VIC-1541

SINGLE DRIVE FLOPPY DISK

USER'S MANUAL

commodore
COMPUTER

INFORMATION TO USER

"WARNING: THIS EQUIPMENT HAS BEEN CERTIFIED TO COMPLY WITH THE LIMITS FOR A CLASS B COMPUTING DEVICE, PURSUANT TO SUBPART J OF PART 15 OF FCC RULES. ONLY PERIPHERALS (COMPUTER INPUT/OUTPUT DEVICES, TERMINALS, PRINTERS, ETC.) CERTIFIED TO COMPLY WITH THE CLASS B LIMITS MAY BE ATTACHED TO THIS COMPUTER. OPERATION WITH NON-CERTIFIED PERIPHERALS IS LIKELY TO RESULT IN INTERFERENCE TO RADIO AND TV RECEPTION."

"THIS EQUIPMENT GENERATES AND USES RADIO FREQUENCY ENERGY AND IF NOT INSTALLED PROPERLY, THAT IS, IN STRICT ACCORDANCE WITH THE MANUFACTURER'S INSTRUCTIONS, MAY CAUSE INTERFERENCE TO RADIO AND TELEVISION RECEPTION. IT HAS BEEN TYPE TESTED AND FOUND TO COMPLY WITH THE LIMITS FOR A CLASS B COMPUTING DEVICE IN ACCORDANCE WITH THE SPECIFICATIONS IN SUBPART J OF PART 15 OF FCC RULES, WHICH ARE DESIGNED TO PROVIDE REASONABLE PROTECTION AGAINST SUCH INTERFERENCE IN A RESIDENTIAL INSTALLATION. HOWEVER, THERE IS NO GUARANTEE THAT INTERFERENCE WILL NOT OCCUR IN A PARTICULAR INSTALLATION. IF THIS EQUIPMENT DOES CAUSE INTERFERENCE TO RADIO OR TELEVISION RECEPTION, WHICH CAN BE DETERMINED BY TURNING THE EQUIPMENT OFF AND ON, THE USER IS ENCOURAGED TO TRY TO CORRECT THE INTERFERENCE BY ONE OR MORE OF THE FOLLOWING MEASURES:

- REORIENT THE RECEIVING ANTENNA
- RELOCATE THE COMPUTER WITH RESPECT TO THE RECEIVER
- MOVE THE COMPUTER AWAY FROM THE RECEIVER
- PLUG THE COMPUTER INTO A DIFFERENT OUTLET SO THAT COMPUTER AND RECEIVER ARE ON DIFFERENT BRANCH CIRCUITS

"IF NECESSARY, THE USER SHOULD CONSULT THE DEALER OR AN EXPERIENCED RADIO/TELEVISION TECHNICIAN FOR ADDITIONAL SUGGESTIONS. THE USER MAY FIND THE FOLLOWING BOOKLET PREPARED BY THE FEDERAL COMMUNICATIONS COMMISSION HELPFUL: 'HOW TO IDENTIFY AND RESOLVE RADIO-TV INTERFERENCE PROBLEMS.' THIS BOOKLET IS AVAILABLE FROM THE U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, D.C. 20402, STOCK NO. 004-000-00345-4."

PART NO: 320970

VIC-1541

SINGLE DRIVE FLOPPY DISK

USER'S MANUAL

P/N 1540031-02

commodore COMPUTER

The information in this manual has been reviewed and is believed to be entirely reliable. No responsibility, however, is assumed for inaccuracies. The material in this manual is for information purposes only, and is subject to change without notice.

Second edition

Table of Contents

1. GENERAL DESCRIPTION

Introduction

Welcome to the fastest, easiest, and most efficient filing system available for your Commodore 64 or VIC 20 computer, your 1541 DISK DRIVE. This manual has been designed to show you how to get the most from your drive, whether you're a beginner or an advanced professional.

If you are a beginner, the first few chapters will help you through the basics of disk drive installation and operation. As your skill and programming knowledge improves, you will find more uses for your disk drive and the more advanced chapters of this manual will become much more valuable.

If you're a professional, this reference guide will show you how to put the 1541 through its paces to perform just about all the disk drive jobs you can think of.

No matter what level of expertise you have, your 1541 disk drive will dramatically improve the overall capabilities of your computer system.

Before you get to the details of 1541 operation, you should be aware of a few important points. This manual is a REFERENCE GUIDE, which means that unless the information you seek directly pertains to the disk or disk drive you will have to use your Commodore 64 or VIC 20 User's Guides and Programmer's Reference Guides to find programming information. In addition, even though we give you step-by-step instructions for each operation, you should become familiar with BASIC and the instructions (called commands) that help you operate your disks and drives. However, if you just want to use your disk drive unit to load and save prepackaged software, we've included an easy and brief section on doing just that.

Now . . . let's get on with the general information.

The commands for the disk drive come in several levels of sophisication. Starting in chapter three, you can learn how the commands that allow you to save and load programs with the disk work. Chapter four teaches you how commands are sent to the disk, and introduces the disk maintenance commands.

Chapter five tells you how to work with sequential data files. These are very similar to their counterparts on tape (but much faster). Chapter six introduces the commands that allow you to work with random files, access any piece of data on the disk, and how you organize the diskette into tracks and blocks. Chapter seven describes the special relative files. Relative files are the best method of storing data bases, especially when they are used along with sequential files.

Chapter eight describes methods for programming the disk controller circuits at the machine language level. And the final chapter shows you how to

change the disk device number, by "cutting" a line inside the drive unit or through software.

Remember, you don't really need to learn everything in this book all at once. The first four chapters are enough to get you going, and the next couple are enough for most operations. Getting to know your disk drive will reward you in many ways–speed of operation, reliability, and much more flexibility in your data processing capabilities.

Specifications

This disk drive allows you to store up to 144 different programs and/or data files on a single mini-floppy diskette, for a maximum of over 174,000 bytes worth of information storage.

Included in the drive is circuitry for both the disk controller and a complete disk operating system, a total of 16K of ROM and 2K of RAM memory. This circuitry makes your Commodore 1541 disk drive an "intelligent" device. This means it does its own processing without taking any memory away from your Commodore 64 or VIC 20 computer. The disk uses a "pipeline" software system. The "pipeline" makes the disk able to process commands while the computer is performing other jobs. This dramatically improves the overall throughput (input and output) of the system.

Diskettes that you create in this disk drive are read and write compatible with Commodore 4040 and 2031 disk drives. Therefore, diskettes can be used interchangeably on any of these systems. In addition, this drive can read programs created on the older Commodore 2040 drives.

The 1541 disk drive contains a dual "serial bus" interface. This bus was specially created by Commodore. The signals of this bus resemble the parallel IEEE-488 interface used on Commodore PET computers, except that only one wire is used to communicate data instead of eight. The two ports at the rear of the drive allows more than one device to share the serial bus at the same time. This is accomplished by "daisy-chaining" the devices together, each plugged into the next. Up to five disk drives and one printer can share the bus simultaneously.

Figure 1.1 Specifications VIC 1540/1541 Single Drive Floppy Disk

STORAGE

Total capacity	174848 bytes per diskette
Sequential	168656 bytes per diskette
Relative	167132 bytes per diskette
	720 ~~65535~~ records per file
Directory entries	144 per diskette
Sectors per track	17 to 21
Bytes per sector	256
Tracks	35
Blocks	683 (664 blocks free)

IC's:

6502	microprocessor
6522 (2)	I/O, internal timers
Buffer	
2114 (4)	2K RAM

PHYSCIAL:

Dimensions	
Height	97 mm
Width	200 mm
Depth	374 mm

Electrical:

Power requirements	
Voltage	100, 120, 220, or 240 VAC
Frequency	50 or 60 Herts
Power	25 Watts

MEDIA:

Diskettes	Standard mini 5¼", single sided, single density

2. UNPACKING AND CONNECTING

Contents of Box

Included with the 1541 disk drive unit, you should find a gray power cable, black serial bus cable, this manual, and a demonstration diskette. The power cable has a connection for the back of the disk drive on one end, and for a grounded (three-prong) electrical outlet on the other. The serial bus cable is exactly the same on both ends. It has a 6-pin DIN plug which attaches to the VIC 20, Commodore 64 or another disk drive.

Please, don't hook up anything until you've completed the following section!

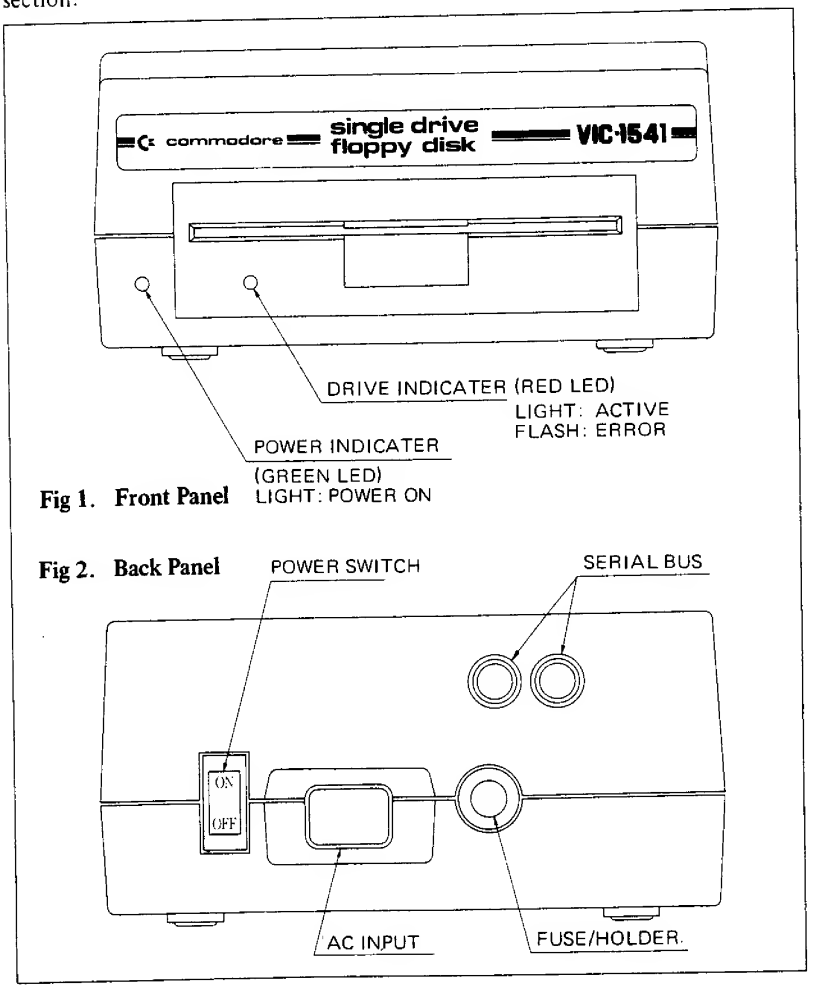


Fig 1. Front Panel

Fig 2. Back Panel

Connection of Cables

Your first step is to take the power cable and insert it into the back of the disk drive (see figure 2.2). It won't go in if you try to put in in upside down. Once it's in the drive, plug the other end into the electrical outlet. **if the disk drive makes any sound at this time, please turn it off using the switch on the back! Don't plug any other cables into the disk drive if the power is on.**

Next, take the serial bus cable and attach it to either one of the serial bus sockets in the rear of the drive. Turn off the computer, and plug the other end of the cable into the back of the computer. That's all there is to it!

If you have a printer, or any additional disk drives, attach the cables into the second serial bus port (see figure 2.3). For directions on using multiple drives at one time, read chapter 9.If you are a first-time user with more than one drive, start working with only one drive until you're comfortable with the unit.

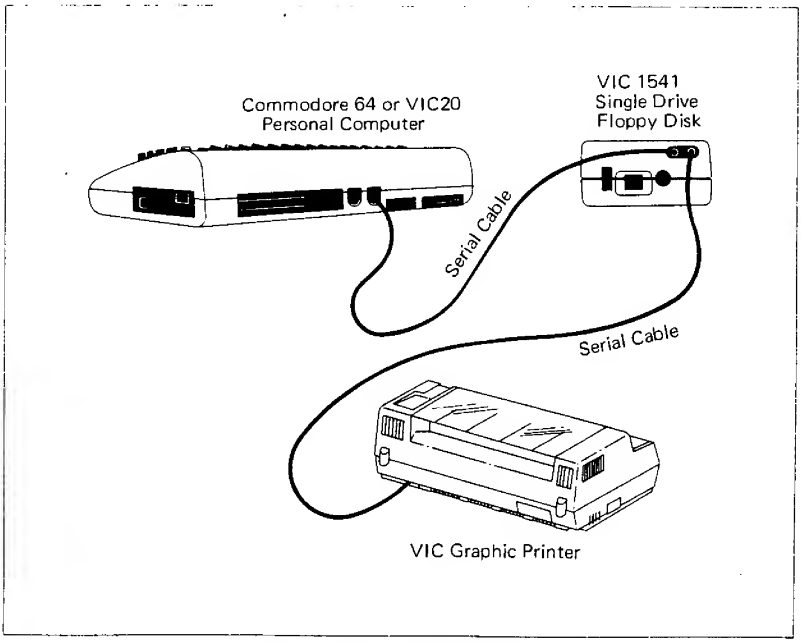


Fig 3. Floppy Disk Hookup

Powering On

When you have all the devices hooked together, it's time to start turning on the power. **It is important that you turn on the devices in the correct order.** The computer should always be turned on last. As long as the computer is the last one to be turned on, everything will be OK.

First, make sure that you've removed all diskettes from the disk drives before powering on.

After all the other devices have been turned on, only then is it safe to turn on the computer. All the other devices will go through their starting sequences. The printer's motor goes on, with the print head moving halfway across the line and back again. The 1541 disk drive will have its red light on, and then the green drive light will blink, while your TV screen forms the starting picture.

Once all the lights have stopped flashing on the drive, it is safe to begin working with it.

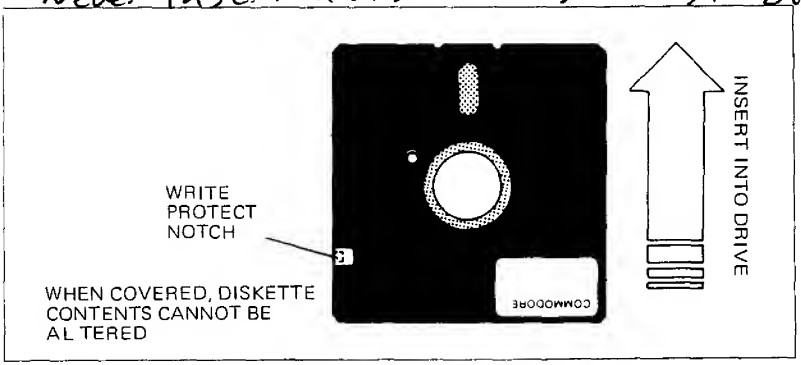


Fig.4. Position for Diskette Insertion

Insertion of Diskette

To open the door on the drive, simply press the door catch lightly, and the door will pop open. If there is a diskette in the drive, it is ejected by a small spring. Take the diskette to be inserted, and place it in the drive face-up with the large opening going in first and the write-protect notch to the left (covered with tape in the demonstration disk) (see figure 2.4).

Press it in gently, and when the diskette is in all the way, you will feel a click and the diskette will not spring out. Close the drive door by pulling downward until the latch clicks into place. Now you are ready to begin working with the diskette.

Remember to always remove the diskette before the drive is turned off or on. Never remove the diskette when the green drive light is on! Data can be destroyed by the drive at this time!

Using With a VIC 20 or Commodore 64

The 1541 Disk Drive can work with either the VIC 20 or Commodore 64

computers. However, each computer has different requirements for speed of incoming data. Therefore, there is a software seitch for selecting which computer's speed to use. The drive starts out ready for a Commodore 64. To switch to VIC 20 speed, the following command must be sent after the drive is started (power-on or through software):

OPEN 15, 8, 15, "UI–": CLOSE 15

To return the disk drive to Commodore 64 speed, use this command:

OPEN 15, 8, 15, "UI+": CLOSE 15

More about using this type of command is in chapter 4, with a detailed explanation of the U (user) commands in chapter 8.

3. USING PROGRAMS

LOADING PREPACKAGED PROGRAMS

For those of you interested in using only prepackaged programs available on cartridges, cassette, or disk, here's all you have to do:

Using your disk drive, carefully insert the preprogrammed disk so that the label on the disk is facing up and is closest to you. Look for a little notch on the disk (it might be covered with a little piece of tape). If you're inserting the disk properly, the notch will be on the left side. Once the disk is inside, close the protective gate by pushing in on the lever. Now type LOAD "PROGRAM NAME", 8 and hit the **RETURN** key. The disk will make noise and your screen will say:

```
SEARCHING FOR PROGRAM NAME
LOADING

READY
■
```

When the READY comes on and the ■ is on, just type RUN, and your prepackaged software is ready to use.

LOAD

The BASIC commands used with programs on the disk drive are the same as the commands used on the Commodore Datassette™ recorder. There are a few extra commands available for use with disks, however. First of all, the program name must be given with each command. On a Datassette, you could omit the program name in order to just LOAD the firs program there. On disk, since there are many different programs that are equally accessible, the program

name must be used to tell the disk drive what to do. In addition, the disk drive's device number must be specified. If no device number is listed, the computer assumes the program is on tape.

FORMAT FOR THE **LOAD** COMMAND:

LOAD name$, device# , command#

The program name is a string, that is, either a name in quotes or the contents of a given string variable. Some valid names are: "HELLO", "PROGRAM #1", A$, NAME$.

The device# is preset on the circuit board to be #8. If you have more than one drive, see chapter 9 on how to change the device number. This book assumes that you're using device number 8 for the disk drive.

The command# is optional. If not given, or zero, the program is LOADed normally, that is, beginning at the start of your available memory for BASIC programs. If the number is 1, the program will be LOADed at **exactly the same memory locations from which it came.** In the case of computers with different memory configurations, like VICs with 5K, 8K, or more memory, the start of BASIC memory is in different places. The command# 0 permits BASIC programs to LOAD normally. Command# 1 is used mainly for machine language, character sets, and other memory dependent functions.

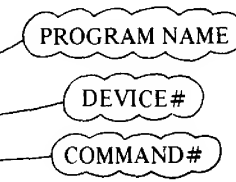
EXAMPLES

LOAD "TEST", 8

LOAD "Program # 1", 8

LOAD A$, J , K

LOAD "Mach Lang", 8, 1



NOTE: You can use variables to represent device numbers, commands, and strings, as long as you've previously defined them in your program.

Directory of Diskette

Your Datassette™ tape deck is a sequential device. It can only read from the beginning of the tape to the end, without skipping around the tape and without the capability of going backward or recording over old data.

Your disk drive is a random access device. The read/write head of the disk can go to any spot on the disk and access a single block of data which holds up to 256 bytes of information. There are a total of 683 blocks on a single diskette.

Fortunately, you don't really have to worry about individual blocks of data. There is a program in the disk drive called the Disk Operating System, or the DOS. This program keeps track of the blocks for you. It organizes them into a Block Availability Map, or BAM, and a directory.

The Block Availability Map is simply a checklist of all 683 blocks on the disk. It is stored in the middle of the diskette, halfway between the center hub and the outer rim. Every time a program is SAVEd or a data file is CLOSEd, the BAM is updated with the list of blocks used up.

The directory is a list of all programs and other files stored on the disk. It is physically located right next to the BAM. There are 144 entries available in the directory, consisting of information like file name and type, a list of blocks used, and the starting block. The directory is automatically updated every time a program is SAVEd or a file is OPENed for writing. Beware: the BAM isn't updated until the file is CLOSEd, even though the directory changes right away. If a file isn't CLOSEd properly, all data in that file will probably be lost.

The directory can be LOADed into your memory just like a BASIC program. Place the diskette in the drive, and type the following command:

```
LOAD "$", 8
```

The computer responds with:

```
SEARCHING FOR $

FOUND $

LOADING

READY.
```

Now the directory is in your computer's memory. Type LIST, and you'll see the directory displayed on the screen. To print the directory on your printer, type the following command line (in this example your printer is plugged in as device# 4):

```
OPEN 4, 8, 4: CMD 4: LIST
```

NOTE: When using CMD, the file must be closed using the command PRINT# 4: CLOSE 4. See the VIC 1525/1515 printer manual for detailed explanation.

To read the directory **without** LOADing it into your memory, see the section later in this chapter on the DOS Support Program. In addition, to examine the directory from inside a BASIC program, see the section in chapter 5 that deals with the GET# statement.

Pattern Matching and Wild Cards

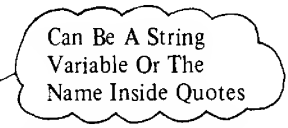
When using the tape deck, you can LOAD any program starting with certain letters just by leaving off any following letters. Thus, the command

LOAD "T" will find the first program on the tape beginning with the letter T. And LOAD "HELLO" will find the first program beginning with the letters HELLO, like "HELLO THERE."

When using the disk, this option is called **pattern matching**, and there is a special character in the file name used to designate this. The asterisk (*) character following any program name tells the drive you want to find any program starting with that name.

FORMAT FOR PATTERN MATCHING:

LOAD name$ + "*", 8



In other words, if you want to LOAD the first program on the disk starting with the letter T, use the command LOAD "T*", 8.

If only the "* is used for the name, the last program accessed on the disk is the one LOADed. If no program has yet been LOADed, the first one listed in the directory is the one used.

You are probably familiar with the concept of wild cards in poker where one card can replace any other card needed. On your 1541, the question mark (?) can be used as a wild card on the disk. The program name on the disk is compared to the name in the LOAD command, but any characters where there is a question mark in the name aren't checked.

For instance, when the command LOAD "T?NT", 8 is given, programs that match include TINT, TENT, etc.

When LOADing the directory of the disk, pattern matching and wild cards can be used to check for a list of specific programs. If you gave the command LOAD "$0:TEST", only the program TEST would appear in the directory (if present on the disk). The command LOAD "$0:t*" would give you a directory of all programs beginning with the letter T. And LOAD "$0:T?ST" would give you all the programs with 4-letter names having the first letter of T and the third and fourth letters ST. LOAD "$0:T?ST*" would give names of any length with the correct first, third, and fourth letters.

Every character after a * is ignored

SAVE

To SAVE a program to the diskette, all that is needed is to add the device number after the program name. Just like the SAVE command for the tape deck, the device number can be followed by a command number, to prevent the automatic re-location on LOADing (see the section on the LOAD command, above).

FORMAT FOR THE SAVE COMMAND:

SAVE name$, device#, command#

↑ ignored by disk drive

See the LOAD command (page 10) for an explanation of the parameters device # and command #.

When you tell the disk drive to SAVE a program, the DOS must take several steps. First, it looks at the directory to see if a program with that name already exists. Next it checks to see that there is a directory entry available for the name. Then it checks the BAM to see if there are enough blocks in which to store the program. If everything is OK up to this point, the program is stored. If not, the error light will flash.

SAVE and Replace

If a program already exists on the disk, it is often necessary to make a change and re-SAVE it onto the disk. In this case, it would be inconvenient to have to erase the old version of the program and then SAVE it.

If the first characters of the program name are the "@" sign followed by a 0 and a colon (:), the DOS knows to replace any old program that has that name with the program that is now in the computer's memory. The drive checks the directory to find the old program, then it marks that entry as deleted, and it next creates a new entry with the same name. Finally, the program is stored normally.

FORMAT FOR SAVE WITH REPLACE:

SAVE "@0:"+ name$, device#, command#

For example, if a file was called TEST, the SAVE and replace command would be SAVE "@0:TEST",8.

no space unless it is to be part of file name

The reason for the 0: is to keep compatibility with other Commodore disk drive units which have more than one drive built in. In that case, the number 0 or 1 is used to specify which drive is being used.

VERIFY

The VERIFY command works to check the program currently in memory against the program on disk. You must include a device# with the VERIFY command. The computer does a byte-by-byte comparison of the program, including line links—which may be different for different memory configurations. For instance, if a program was SAVEd to disk from a 5K VIC 20, and re-LOADed on an 8K machine, it wouldn't VERIFY properly because the links point to different memory locations.

FORMAT FOR VERIFY COMMAND:

VERIFY name$, device#

DOS Support Program

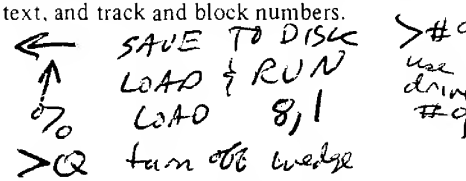
On your demonstration disk, there may be a program called DOS SUPPORT. This program, also called a wedge, allows you to use many disk commands more easily (different wedges are used for the VIC 20 and the Commodore 64). Just LOAD the program and RUN it. It automatically sets itself up and erases itself when it's finished. You'll have a few hundred less bytes to work with when this program is running, but you'll also have a handy way to send the disk commands.

As a result of the DOS Support, the "/" key now takes the place of the LOAD command. Just hit the slash followed by the program name, and the program is LOADed. When you use this method, you don't need to use the LOAD command or the comma 8.

The "@" and ">" keys are used to send commands to the disk drive. If you type @$ (or >$), the directory of the disk is displayed on the screen, without LOADing into your memory! These keys also take the place of the PRINT# (see chapter 4) to send commands listed in the next chapter.

To read the error channel of the disk (when the red error light is blinking), just hit either the @ or the > and hit RETURN. The complete error message is displayed to you: message number, text, and track and block numbers.

4: DISK COMMANDS



OPEN and PRINT#

Up 'til now, you have explored the simple ways of dealing with the disk drive. In order to communicate with the disk drive more fully, you have to touch on the OPEN and PRINT# statements in BASIC (more details of these commands are available in your VIC 20 or Commodore 64 User's Guide or Programmer's Reference Guide). You may be familiar with their use with data files on cassette tape, where the OPEN statement creates the file and the PRINT# statement fills the file with data. They can be used the same way with the disk, as you will see in the next chapter. But they can also be used to set up a command channel. The command channel lets you exchange information between the computer and the disk drive.

FORMAT FOR THE OPEN STATEMENT:

OPEN file#, device#, channel#, text S

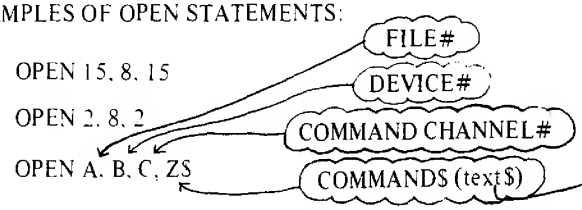
The file# can be any number from 1 to 255. This number is used throughout the program to identify which file is being accessed. But numbers greater than 127 should be avoided, because they cause the PRINT# statement to generate a linefeed after the return character. These numbers are really meant to be used with non-standard printers.

The device# of the disk is usually 8. 0 to 31

The channel# can be any number from 2 to 15. These refer to a channel used to communicate with the disk, and channels numbered 0 and 1 are reserved for the operating system to use for LOADing and SAVEing. Channels 2 through 14 can be used for data to files, and 15 is the **command channel.**

The text$ is a string that is PRINTed to the file, as if with a PRINT# statement. This is handy for sending a single command to the channel.

EXAMPLES OF OPEN STATEMENTS:



The PRINT# command works **exactly** like a PRINT statement, except that the data goes to a device other than the screen, in this case to the disk drive. When used with a data channel, the PRINT# sends information into a buffer in the disk drive, from which it goes to the diskette. When PRINT# is used with the command channel, it sends commands to the disk drive.

FORMAT FOR SENDING DISK COMMANDS:

```
OPEN 15, 8, 15, command$
```

or

```
PRINT# 15, command$
```

NEW

This command is necessary when using a diskette for the first time. The NEW command erases the entire diskette, it puts timing and block markers on the diskette and creates the directory and BAM. The NEW command can also be used to clear out the directory of an already-formatted diskette. This is faster than re-formatting the whole disk.

FORMAT FOR THE NEW COMMAND TO FORMAT DISK:

doesn't seem to give an error msg — do a → $ to clear format

```
PRINT#15, "NEWØ:name,id"
```
DRIVE#

or abbreviated as

```
PRINT#15, "NØ:name,id"
```

FORMAT FOR THE NEW COMMAND TO CLEAR DIRECTORY:

```
PRINT# 15, "NØ:name"
```

The name goes in the directory as the name of the entire disk. This only appears when the directory is listed. The ID code is any 2 characters, and they are placed not only on the directory but on every block throughout the diskette.

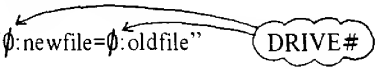
That way, if you carelessly replace diskettes while writing data, the drive will know by checking the ID that something is wrong.

COPY

This command allows you to make a copy of any program or file on the disk drive. It won't copy from one drive to a different one (except in the case of dual drives like the 4040), but it can duplicate a program under another name on the drive.

FORMAT FOR THE COPY COMMAND:

PRINT# 15, "COPY∅:newfile=∅:oldfile"



or abbreviated as

PRINT# 15, "c∅:newfile=∅:oldfile"

The COPY command can also be used to combine two through four files on the disk.

FORMAT FOR COPY TO COMBINE FILES:

PRINT#15, "C0:newfile=0:oldfile1,0:oldfile2,0:oldfile3,0:oldfile4"

EXAMPLES OF COPY COMMAND:

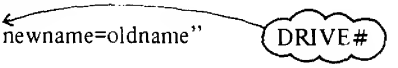
PRINT# 15, "C0:BACKUP=0:ORIGINAL"
PRINT#15, "C0:MASTERFILE=0:NAME,0:ADDRESS,0:PHONES"

RENAME

This command allows you to change the name of a file once it is in the disk directory. This is a fast operation, since only the name in the directory must be changed.

FORMAT FOR RENAME COMMAND:

PRINT# 15, "RENAME∅:newname=oldname"



or abbreviated as

PRINT# 15, "R∅:newname=oldname"

EXAMPLE OF RENAME COMMAND:

PRINT# 15, "R0:MYRA=MYRON"

The RENAME command will not work on any files that are currently OPEN.

SCRATCH

This command allows you to erase unwanted files and programs from the disk, which then makes the blocks available for new information. You can erase programs one at a time or in groups by using **pattern matching** and/or wild cards.

FORMAT FOR SCRATCH COMMAND

PRINT# 15, "SCRATCHØ:name" (DRIVE#)
or abbreviated as
PRINT# 15, "SØ:name"

If you check the error channel after a scratch operation (see below), the number usually reserved for the track number now tells you how many files were scratched. For example, if your directory contains the programs KNOW and GNAW, and you use the command PRINT# 15, "S0:?N?W", you will scratch **both** programs. If the directory contains TEST, TRAIN, TRUCK, and TAIL, and you command the disk to PRINT# 15, "S0:T*", you will erase all four of these programs.

S0: name1, name2 also works

INITIALIZE

At times, an error condition on the disk will prevent you from performing some operation you want to do. The INITIALIZE command returns the disk drive to the same state as when powered up. You must be careful to re-match the drive to the computer (see chapter 2).
FORMAT FOR INITIALIZE COMMAND:

PRINT# 15, "INITIALIZE"
or abbreviated as
PRINT# 15, "I"

VALIDATE

After a diskette has been in use for some time, the directory can become disorganized. When programs have been repeatedly SAVEd and SCRATCHed, they may leave numerous small gaps on the disk, a block here and a few blocks there. These blocks never get used because they are too small to be useful. The VALIDATE command will go in and re-organize your diskette so that you can get the most from the available space.

Also, there may be data files that were OPENed but never properly CLOSEd. This command will collect all blocks taken by such files and make them available to the drive, since the files are unusable at that point.

There is a danger in using this command. When using random files (see chapter 6), blocks allocated will be de-allocated by this command. Therefore, this command should never be used with a diskette that uses random files.

FORMAT FOR VALIDATE COMMAND:

PRINT# 15, "VALIDATE"
or abbreviated as
PRINT# 15, "V"

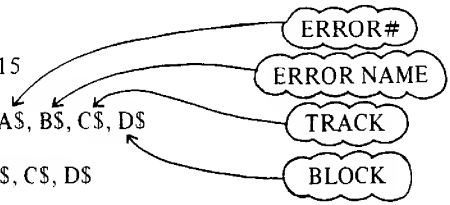
DUPLICATE

This command is a hangover from the operating systems that were contained on the dual drives like the 4040. It was used to copy entire diskettes from one drive to another, but has no function on a single disk drive.

Reading the Error Channel

Without the DOS Support Program, there is no way to read the disk error channel without a program, since you need to use the INPUT# command which won't work outside a program. Here is a simple BASIC routine to read the error channel:

```
10 OPEN 15, 8, 15
20 INPUT# 15, A$, B$, C$, D$
30 PRINT A$, B$, C$, D$
```



Whenever you perform an INPUT# operation from the command channel, you read up to 4 variables that describe the error condition. The first, third, and fourth variables come in as numbers, and can be INPUT into numeric variables if you like. The first variable describes the error #, where 0 is no error. The second variable is the error description. The third variable is the track number on which the error occurred, and the fourth and final is the block number inside that track. (A block is also known as a sector)

Errors on track 18 have to do with the BAM and directory. For example, a READ ERROR on track 18 block 0 may indicate that the disk was never formatted.

CLOSE

It is extremely important that you properly CLOSE files once you are finished using them. Closing the file causes the DOS to properly allocate blocks in the BAM and to finish the entry in the directory. **If you don't CLOSE the file, all your data will be lost!**

FORMAT FOR CLOSE STATEMENT:

CLOSE file#

You should also be careful not to CLOSE the error channel (channel# 15)

before CLOSEing your data channels. **The error channel should be OPENed first and CLOSEd last of all your files!** That will keep your programs out of trouble.

If you close the error channel while other files are OPEN, the disk drive will CLOSE them for you, but BASIC will still think they are open (unless you CLOSE them properly), and let you to try to write to them.

NOTE: If your BASIC program leads you into an error condition, all files are CLOSEd in BASIC, **without** CLOSEing them on your disk drive! This is a **very dangerous** condition. You should immediately type the statement OPEN 15, 8, 15, "I". This will re-initialize your drive and make all your files safe.

5. SEQUENTIAL FILES

OPEN

Sequential files on the disk drive work exactly like they do on cassette tape, only much faster. They are limited by their sequential nature, which means they must be read from beginning to end. Data is transferred byte by byte, through a buffer, onto the magnetic media. To the disk drive all files are created equal. That is, sequential files, program files, and user files all work the same on the disk. Only program files can be LOADed, but that's really the only difference. Even the directory works like this, except that it is read-only. The only difference is with relative files.

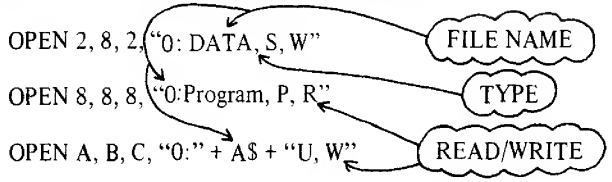
FORMAT FOR OPENING A SEQUENTIAL FILE:

OPEN file#, device#, channel#, "0:name,type,direction"

The file number is the same as in all your other applications of the OPEN statement, and it is used throughout the program to refer to this particular file. The device# is usually 8. The channel# is a data channel, number 2 through 14. It is convenient to use the same number for both the channel# and file#, to keep them straight. The name is the file name (no wild cards or pattern matching if you're creating a write file). The type can be any of the ones from the chart below, at least the first letter of each type. The direction must be READ or WRITE, or at least the first letter of each.

FILE TYPE	MEANING
PRG	Program
SEQ	Sequential
USR	User
REL	Relative (not implemented in BASIC 2.0)

EXAMPLES OF OPENING SEQUENTIAL FILES:



If the file already exists, you can use the **replace** option in the OPEN statement, similar to the SAVE-and-replace described in chapter 3. Simply add the @0: before the file's name in the OPEN statement.

EXAMPLE OF SEQUENTIAL FILE WITH REPLACE OPTION:

```
OPEN 2, 8, 2, "@0:DATA,S,W"
```

PRINT# and INPUT#

The PRINT# command works **exactly** like the PRINT statement, except that output is re-directed to the disk drive. The reason for the special emphasis on the word exactly is that all the formatting capabilities of the PRINT statement, as applies to punctuation and data types, applies here too. It just means that you have to be careful when putting data into your files.

FORMAT FOR WRITING TO FILE WITH PRINT#:

```
PRINT# file#, data list
```

The file# is the one from the OPEN statement when the file was created.

The data list is the same as the regular PRINT statement--a list of variables and/or text inside quote marks. However, you must be especially careful when writing data that it is as easy as possible to read back again later.

When using the PRINT# statement, if you use commas (,) to separate items on the line, the items will be separated by some blank spaces, as if it were being formatted for the screen. Semicolons (;) don't result in any extra spaces.

In order to more fully understand what's happening, here is a diagram of a sequential file created by the statement OPEN 5, 8, 5, "0:TEST,S,W":

	eof															
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...

The **eof** stands for the **end-of-file** marker. String data entering the file goes in byte by byte, including spaces.

For instance, let's set up some variables with the statement A$= "HELLO"; B$= "ALL": C$= "BYE". Here is a picture of a file after the

statement PRINT# 5, A$; B$; C$:

	H	E	L	L	O	A	L	L	B	Y	E	CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13

CR stands for the **CHR$ code of** 13, the **carriage return,** which is PRINTed at the end of every PRINT or PRINT# statement unless there is a comma or semicolon at the end of the line.

NOTE: Do not leave a space between PRINT and #, and do not try to abbreviate the command as ?#. See the appendixes in the user manual for the correct abbreviation.

FORMAT FOR INPUT# STATEMENT:

INPUT# file#, variable list

When using the INPUT# to read this data in, there is no way to tell that it's not supposed to be one long string. You need something in the file to act as a separator. Characters to use as separators include the CR, a comma or a semicolon. The CR can be added easily by just using one variable per line on the PRINT# statement, and the system puts one there automatically. The statement PRINT# 5, A$: PRINT# 5, B$: PRINT# 5, C$ puts a CR after every variable being written, providing the proper separation for a statement like INPUT#5, A$, B$, C$. Or else a line like Z$= ",": PRINT# 5, A$ Z$ B$ Z$ C$ will do the job as well, and in less space. The file after that line looks like this:

	H	E	L	L	O	,	A	L	L	,	B	Y	E	CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Putting commas between variables results in lots of extra space on the disk being used. A statement like PRINT# 5, A$, B$ makes a file that looks like:

	H	E	L	L	O							A	L	L		CR	eof
char	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	23	24

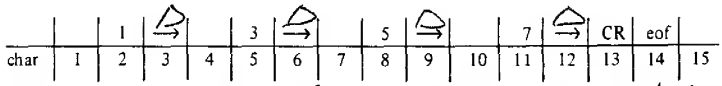
You can see that much of the space in the file is wasted.

The moral of all this is: take care when using PRINT# so your data will be in order for reading back in.

Numeric data written in the file takes the form of a string, as if the STR$ function had been performed on it before writing it out. The first character will be a blank space if the number is positive, and a minus sign (-) if the number is negative. Then comes the number,and the last character is the ~~cursor right~~ space character. This format provides enough information for the INPUT# statement to read them in as separate numbers if several are written with no other special separators. It is somewhat wasteful of space, since there can be two unused

characters if the numbers are positive.

Here is a picture of the file after the statement PRINT# 5, 1; 3; 5; 7 is performed:



To omit spaces after #: PRINT# 2, STR$(a)";

Appendix B contains a program demonstrating the use of a sequential disk file. To omit space before +#: MID$(STR$(a),2);

GET#

The GET# retrieves data from the disk, one character at a time.

FORMAT FOR THE GET# STATEMENT:

GET# file#, variable list

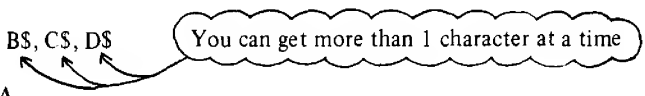
Data comes in byte by byte, including the CR, comma, and other separating characters. It is much safer to use string variables when using the GET# statement. You will get a BASIC error message if string data is received where a number was requested, but not vice-versa.

EXAMPLES OF GET# STATEMENT:

GET# 5, A$

GET# A, B$, C$, D$

GET$ 5, A



The GET# statement is extremely useful when examining files with unknown contents, like a file that may have been damaged by an experimental program. It is safer than INPUT# because there is a limit to the number of characters allowed between separators of INPUT variables. With GET#, you receive every character, and you can examine separators as well as other data.

Here is a sample program that will allow you to examine any file on the disk:

```
10 INPUT "FILE NAME"; F$
20 INPUT "FILE TYPE";T$
30 T$=LEFT$(T$,1)
40 IF T$ <> "S" THEN IF T$ <> "P" THEN IF T$ <> "U" THEN 20
45 OPEN 15, 8, 15
50 OPEN 5, 8, 5, "0:" + F$+ ","+ T$+ ",R"
60 GOSUB 200
```

```
 70 GET# 5, A$
 80 IF ST <> 0 THEN PRINT ST: STOP
 90 PRINT ASC(A$+CHR$(0) );
100 GOTO 70
200 INPUT# 15, A$, B$, C$, D$
210 IF VAL (A$) > 0 THEN PRINT A$,B$,C$;D$:STOP
220 RETURN
```

In Case Of Null Character Being Read In – Causes Error With ASC Function Otherwise!

Reading the Directory

The directory of the diskette may be read just like a sequential file. Just use $ for the file name, and OPEN 5, 8, 5, "$". Now the GET# statement works to examine the directory. The format here is identical to the format of a program file: the file sizes are the line numbers, and names are stored as characters within quote marks.

Here's a program that lets you read the directory of the diskette:

```
 10 OPEN1,8,2,"$"
 20 GET#1,A$,A$,A$,A$
 30 T$(0) = "Del":T$(1) = "SEQ":T$(2) = "PRG":T$(3) = "USR":T$(4) = "REL"
 40 J=17:GOSUB500
 50 N$=B$
 60 J=2
 70 GOSUB500
 80 I$=B$
 90 J=2
100 GOSUB500
110 O$=B$
120 FORL=1TO73
130 GET#1,A$,A$,A$
140 NEXT
150 GET#1,A$,A$,A$,A$,A$
160 PRINTCHR$(147) "Disk name: "N$,"ID: "I$,"OS: "O$
161 PRINT"Length","Type","Name"
165 FORP=1TO8
170 GET#1,T$,A$,A$
175 IFSTTHENCLOSE1:END
180 IFT$=""THENT$=CHR$(128)
190 J=15
200 GOSUB500
210 N$=B$
220 GET#1,A$,A$,A$,A$,A$,A$,A$,A$,A$,A$,L$,H$
225 L=ASC(L$+CHR$(0))+256*ASC(H$+CHR$(0):IFL=0THEN250
230 PRINTL,T$(ASC(T$)-128),N$
250 IFP < 8THENGET#1,A$,A$
260 NEXTP:GOTO165
```

DISK NAME

ID

OPERATING SYSTEM

GET REST OF BLOCK

FILE NAME

GET FILE LENGTH

```
500 B$=""
510 FORL=0TOJ
520 GET#1,A$
530 IFA$ <> CHR$(96)THENIFA$ <> CHR$(160)THENB$=B$+A$
540 NEXT
550 RETURN
```

BUILD A STRING SUBROUTINE

Table 5.1: 1540/1541 BAM FORMAT

Track 18, Sector 0.		
BYTE	CONTENTS	DEFINITION
0,1	18,01	Track and sector of first directory block
2	65	ASCII character A indicating 4040 format.
3	0	Null flag for future DOS use.
4–143		*Bit map of available blocks for trace 1–35.
*1 = available block 0 = block not available (each bit represents one block)		

Table 5.2: 1540/1541 DIRECTORY HEADER

Track 18, Sector 0.		
BYTE	CONTENTS	DEFINITION
144–161		Disk name padded with shifted spaces.
162–163		Disk ID.
164	160	Shifted space.
165,166	50,65	ASCII representation for 2A which is DOS version and format type.
166–167	160	Shifted spaces.
171–255	0	Nulls, not used.
Note: ASCII characters may appear in locations 180 thru 191 on some diskettes.		

Table 5.3: DIRECTORY FORMAT

Track 18, Sector 1 for 4040 Track 39, Sector 1 for 8050	
BYTE	DEFINITION
0,1	Track and sector of next directory block.
2–31	*File entry 1
34–63	*File entry 2
66–95	*File entry 3
98–127	*File entry 4
130-159	*File entry 5
162–191	*File entry 6
194–223	*File entry 7
226–255	*File entry 8

*STRUCTURE OF SINGLE DIRECTORY ENTRY

BYTE	CONTENTS	DEFINITION
0	128+type	File type OR'ed with $80 to indicate properly closed file. TYPES: 0 = DELeted 1 = SEQential 2 = PROGram 3 = USER 4 = RELative
1,2		Track and sector of 1st data block.
3–18		File name padded with shifted spaces.
19,20		Relative file only: track and sector for first side sector block.
21		Relative file only: Record size.
22–25		Unused.
26,27		Track and sector of replacement file when OPEN@ is in effect.
28,29		Number of blocks in file: low byte, high byte.

Table 5.4: SEQUENTIAL FORMAT

BYTE	DEFINITION
0,1	Track and sector of next sequential data block.
2–255	254 bytes of data with carriage returns as record terminators.

Table 5.5: PROGRAM FILE FORMAT

BYTE	DEFINITION
0,1	Track and sector of next block in program file.
2–255	254 bytes of program info stored in CBM memory format (with key words tokenized). End of file is marked by three zero bytes.

if byte 0 = 0 ⇒ last block
byte 1 = index to last byte

6. RANDOM FILES

Sequential files are fine when you're just working with a continuous stream of data, but some jobs require more than that. For example, with a large mailing list, you would not want to have to scan through the entire list to find a person's record. For this you need some kind of **random access** method, some way to get to any record inside a file without having to read through the entire file first.

There are actually two different types of random access files on the Commodore disk drive. The relative files discussed in the next chapter are more convenient for data handling operations, although the random files in this chapter have uses of their own, especially when working with machine language.

Random files on the Commodore disk drive reach the individual 256-byte blocks of data stored on the disk. As was mentioned in the first chapter, there are a total of 683 blocks on the diskette, of which 664 are free on a blank diskette. Each block of data really means 1 Track and sector of the same name.

The diskette is divided into tracks, which are laid out as concentric circles on the surface of the diskette. There are 35 different tracks, starting with track 1 at the outside of the diskette to track 35 at the center. Track 18 is used for the directory, and the DOS fills up the diskette from the center outward.

Each track is subdivided into sectors. Because there is more room on the outer tracks, there are more sectors there The outer tracks contain 21 sectors each, while the inner ones only have 17 blocks each. The table below shows the number of sectors per track.

Table 6.1: TRACK AND BLOCK FORMAT

TRACK NUMBER	SECTOR RANGE	TOTAL SECTORS
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17

The DOS contains commands for reading and writing directly to any track and sector on the diskette. There are also commands for checking to see which blocks (tracks & sectors) are available, and for marking off used blocks.

These commands are transmitted through the command channel (channel# 15), and tell the disk what to do with the data. The data must be read later through one of the open data channels.

Opening a Data Channel for Random Access

When working with random access files, you need to have 2 channels open to the disk: one for the commands, and the other for the data. The command channel is OPENed to channel 15, just like other disk commands you've encountered so far. The data channel for random access files is OPENed by selecting the pound sign (#) as the file name.

FORMAT FOR OPEN STATEMENT FOR RANDOM ACCESS DATA:

OPEN file#, device#, channel#, "#"
or optionally
OPEN file#, device#, channel#, "# buffer#"

EXAMPLES OF OPENING RANDOM ACCESS DATA CHANNEL:

OPEN 5, 8, 5, "#" — DON'T CARE WHICH BUFFER

OPEN A, B, C, "#2" — PICK BUFFER #2

BLOCK-READ

FORMAT FOR BLOCK-READ COMMAND:

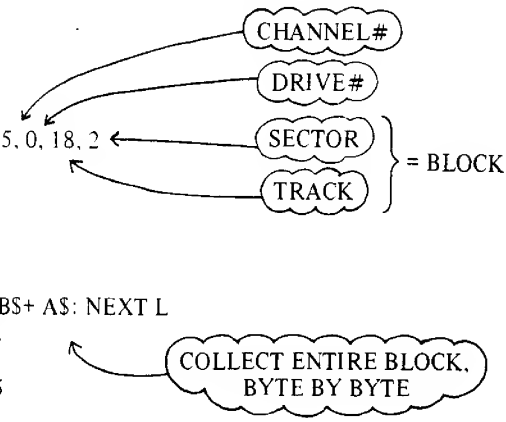
PRINT# file#, "BLOCK-READ:" channel, drive, track, block
or abbreviated as
PRINT# file#, "B-R:" channel, drive, track, block

This command will move one block of data from the diskette into the selected channel. Once this operation has been performed, the INPUT# and GET# statements can read the information.

B-R reads (at most) bytes 1-255 of the block, with byte 0 giving the index of the last byte

SAMPLE PROGRAM TO READ BLOCK 2 FROM TRACK 18: (stores contents in B$)

```
10 OPEN 15, 8, 15
20 OPEN 5, 8, 5, "#"
30 PRINT# 15, "B-R:" 5, 0, 18, 2
40 B$=" "
50 FOR L=0 TO 255
60 GET# 5, A$
70 IF ST=0 THEN B$= B$+ A$: NEXT L
80 PRINT "FINISHED"
90 CLOSE 5: CLOSE 15
```



BLOCK-WRITE

The BLOCK-WRITE command is the exact opposite of the BLOCK-READ command. First you must fill up a data buffer with your information, then you write that buffer to the correct location on the disk.

FORMAT FOR BLOCK-WRITE COMMAND:

PRINT# file#, "BLOCK-WRITE:" drive, channel, track, block
or abbreviated as
PRINT# file, "B-W:" drive, channel, track, block

When the data is being put into the buffer, a pointer in the DOS keeps track of how many characters there are. When you perform the BLOCK-WRITE operation, that pointer is recorded on the disk. That is the reason for the ST check in line 70 of the program above: the ST will become non-zero when you try to read past the end-of-file marker within the record.

SAMPLE PROGRAM TO WRITE DATA ON TRACK 1, SECTOR 1:

```
10 OPEN 15, 8, 15
20 OPEN 5, 8, 5, "#"
30FOR L=1 to 50
40 PRINT#5, "TEST"
50 NEXT
60 PRINT# 15, "B-W:" 5, 0, 1, 1
70 CLOSE 5: CLOSE 15
```

BLOCK-ALLOCATE

In order to safely use random files along with regular files, your programs must check the BAM to find available blocks, and change the BAM to reflect that you've used them. Once you update the BAM, your random files will be safe–at least unless you perform the VALIDATE command (see chapter 3).

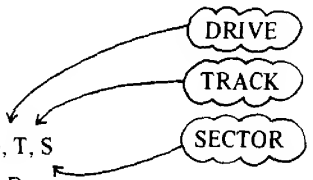
FORMAT FOR THE BLOCK-ALLOCATE COMMAND:

PRINT# file#, "BLOCK-ALLOCATE:" drive, track, block

How do you know which blocks are available to use? If you try a block that isn't available, the DOS will set the error message to number 65, NO BLOCK, and set the track and block numbers to the next available track and block number. Therefore, any time you attempt to write a block to the disk, you must first try to allocate that block. If that block isn't available, read the next block available from the error channel and then allocate that block.

EXAMPLE OF PROCEDURE TO ALLOCATE BLOCK:

```
10 OPEN 15, 8, 15
20 OPEN 5, 8, 5, "#
30 PRINT# 5, "DATA"
40 T=1: S=1
50 PRINT#15, "B-A:" 0, T, S
60 INPUT#15, A, B$, C, D
70 IF A=65 THEN T=C: S=D: GOTO 50
80 PRINT# 15, "B-W:" 5, 0, T, S
```



BLOCK-FREE

The BLOCK-FREE command is the opposite of BLOCK-ALLOCATE, in that it frees a block that you don't want to use anymore for use by the system. It is vaguely similar to the SCRATCH command for files, since it doesn't really erase any data from the disk–just frees the entry, in this case just in the BAM.

FORMAT FOR BLOCK-FREE COMMAND:

PRINT# file#, "BLOCK-FREE:" drive, track, block
or abbreviated as
PRINT# file#, "B-F:" drive, track, block

Using Random Files

The only problem with what you've learned about random files so far is

that you have no way of keeping track of which blocks on the disk you used. After all, you can't tell one used block on the BAM from another. You can't tell whether it contains your random file, just part of a program, or even sequential or relative files.

To keep track, the most common method is to build up a sequential file to go with each random file. Use this file to keep just a list of record, track, and block locations. This means that there are 3 channels open to the disk for each random file: one for the command channel, one for the random data, and the other for the sequential data. This also means that there are 2 buffers that you're filling up at the same time!

SAMPLE PROGRAM WRITING 10 RANDOM-ACCESS BLOCKS WITH SEQUENTIAL FILE:

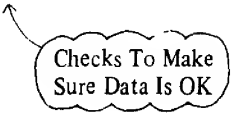
```
 10 OPEN 15, 8, 15
 20 OPEN 5, 8, 5, "#
 30 OPEN 4, 8, 4, "@0:KEYS,S,W"
 40 A$= "RECORD CONTENTS #"
 50 FOR R=1 TO 10
 70 PRINT# 5, A$ "," R
 90 T=1:B =1
100 PRINT# 15, "B-A:" 0, T, B
110 INPUT# 15, A, B$, C, D
120 IF A=65 THEN T=C:B =D: GOTO 100
130 PRINT# 15, "B-W:" 5, 0, T, B
140 PRINT# 4, T ;B
150 NEXT R
160 CLOSE 4: CLOSE 5: CLOSE 15
```

SAMPLE PROGRAM READING BACK 10 RANDOM-ACCESS BLOCKS WITH SEQUENTIAL FILE:

```
 10 OPEN 15, 8, 15
 20 OPEN 5, 8, 5, "#"
 30 OPEN 4, 8, 4, "KEYS,S,R"
 40 FOR R=1 TO 10
 50 INPUT# 4, T, S
 60 PRINT# 15, "B-R:" 5, 0, T, S
```

Finds Next Track & Sector Used

```
 80 INPUT# 5, A$, X
 90 IF A$ <> "Record Contents #" OR X <> R THEN STOP
110 PRINT# 15, "B-F:" 0, T, S
120 NEXT R
130 CLOSE 4: CLOSE 5
140 PRINT# 15, "SØ:KEYS"
150 CLOSE 15
```



BUFFER-POINTER

The buffer pointer keeps track of where the last piece of data was written. It also is the pointer for where the next piece of data is to be read. By changing the buffer pointer's location within the buffer, you can get random access to the individual bytes within a block. This way, you can subdivide each block into records.

For example, let's take a hypothetical mailing list. The information such as name, address, etc., will take up a total of 64 characters maximum. We could divide each block of the random access file into 4 separate records, and by knowing the track, sector, and record numbers, we can access that individual record.

FORMAT FOR BUFFER-POINTER COMMAND:

PRINT# file#, "BUFFER-POINTER:" channel, location
or abbreviated as
PRINT# file#, "B-P:" channel, location

EXAMPLE OF SETTING POINTER TO 64TH CHARACTER OF BUFFER:

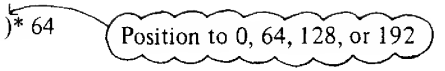
```
PRINT# 15, "B-P:" 5, 64
```

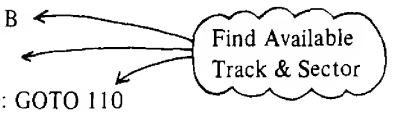
Here are versions of the random access wwiting and reading programs shown above, modified to work with records within blocks:

SAMPLE PROGRAM WRITING 10 RANDOM-ACCESS BLOCKS WITH 4 RECORDS EACH:

```
 10 OPEN 15, 8, 15
 20 OPEN 5, 8, 5, "#
 30 OPEN 4, 8, 4, "KEYS,S,W"
 40 A$= "RECORD CONTENTS #"
 50 FOR R=1 TO 10
 60 FOR L=1 TO 4
```

```
70 PRINT# 15, "B-P:" 5; (L-1)* 64
80 PRINT# 5, A$ "," L
90 NEXT L
100 T=1: B=1
110 PRINT# 15, "B-A:" 0; T; B
120 INPUT# 15, A, B$, C, D
130 IF A=65 THEN T=C: B=D: GOTO 110
140 PRINT# 15, "B-W:" 5; 0; T; B
150 PRINT# 4, T ;B
160 NEXT R
170 CLOSE 4: CLOSE 5: CLOSE 15
```





SAMPLE PROGRAM READING BACK 10 RANDOM-ACCESS BLOCKS WITH 4 RECORDS EACH:

```
10 OPEN 15, 8, 15
20 OPEN 5, 8, 5, "#
30 OPEN 4, 8, 4, "KEYS,S,R"
40 FOR R=1 TO 10
50 INPUT# 4, T, S
60 PRINT# 15, "B-R:" 5; 0; T; S
70 FOR L=1 TO 4
80 PRINT# 15, "B-P:" 5; (L-1)* 64
85 INPUT# 5, A$, X
90 IF A$ <> "Record Contents #" OR X=L THEN STOP
100 NEXT L
110 PRINT# 15, "B-F:" 0; T; S
120 NEXT R
130 CLOSE 4: CLOSE 5
140 PRINT# 15, "S0:KEYS"
150 CLOSE 15
```

USER1 and USER2

The user commands are generally designed to work with machine language (see the next chapter for more on this). The USER1 and USER2 commands are special versions of the BLOCK-READ and BLOCK-WRITE commands, but . . .

with an important difference: the way USER1 and USER2 work with the buffer-pointer.

The BLOCK-READ command reads up to 256 characters, but stops reading when the buffer-pointer stored with the block says that block is finished. The USER1 command performs the BLOCK-READ operation, but first forces the pointer to 255 in order to read the entire block of data from the disk.

FORMAT FOR USER1 COMMAND:

PRINT# file#, "U1:" channel, drive, track, block

or

PRINT# file#, "UA:" channel, drive, track, block

There is no difference between the U1 and UA designations for this command.

The BLOCK-WRITE command writes the contents of the buffer to the block on the disk along with the value of the buffer-pointer. The USER2 command writes the buffer without disturbing the buffer-pointer value already stored on that block of the diskette. This is useful when a block is to be read in with BLOCK-READ, updated through the BUFFER-POINTER and PRINT# statements, and then written back to the diskette with USER2.

FORMAT FOR USER2 COMMAND:

PRINT# file#, "U2:" channel, drive, track, block

or

PRINT# file#, "UB:" channel, drive, track, block

For a more complex sample program, see appendix B.

7. RELATIVE FILES

Relative files allow you to easily zero in on exactly the piece of data that you want from the file. It is more convenient for data handling because it allows you to structure your files into records, and into fields within those records.

The DOS keeps track of the tracks and sectors used, and even allows records to overlap from one block to the next. It is able to do this because it establishes **side sectors,** a series of pointers for the beginning of each record. Each side sector can point to up to 120 records, and there may be 6 side sectors in a file. There can be up to 720 records in a file, and each record can be up to 254 characters, so the file could be as large as the entire diskette.

Creating a Relative File

When a relative file is first to be used, the OPEN statement will create that file; after that, that same file will be used. The replace option (with the @ sign) **does not** erase and re-create the file. The file can be expanded, read, and written into.

FORMAT FOR THE OPEN STATEMENT TO CREATE RELATIVE FILE:

OPEN file#, device#, channel#, "name,L," + CHR$(record length)

EXAMPLES OF OPEN STATEMENT CREATING RELATIVE FILES:

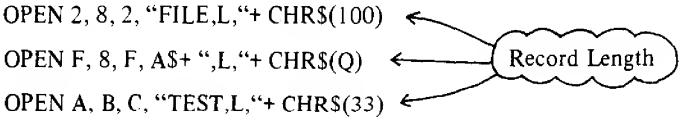


Table 7.1 RELATIVE FILE FORMAT

DATA BLOCK	
BYTE	DEFINITION
0,1	Track and sector of next data block.
2–255	254 bytes of data. Empty records contain FF (all binary ones) in the first byte followed by 00 (binary all zeros) to the end of the record. Partially filled records are padded with nulls (00).
SIDE SECTOR BLOCK	
BYTE	DEFINITION
0,1	Track and sector of next side sector block.
2	Side sector number. (0-5)
3	Record length.
4,5	Track and sector of first side sector (number 0)
6,7	Track and sector of second side sector (number 1)
8,9	Track and sector of third side sector (number 2)
10,11	Track and sector of fourth side sector (number 3)
12,13	Track and sector of fifth side sector (number 4)
14,15	Track and sector of sicth side sector (number 5)
16–256	Track and sector pointers to 120 data blocks.

Upon execution, the DOS first checks to see if the file exists. If it does, then nothing happens. The only way to erase an old relative file is by using the SCRATCH command (see chapter 4), but **not** by using the replace option.

Using Relative Files

In order to OPEN a relative file once it exists, the format is simpler.

FORMAT FOR OPENING AN EXISTING RELATIVE FILE:

OPEN file#, device#, channel#, "name"

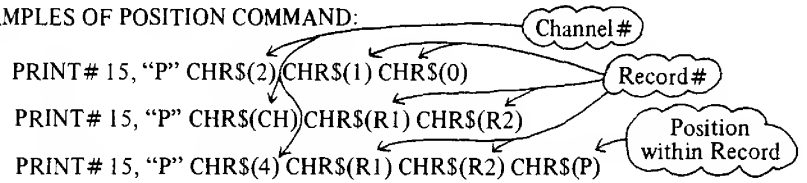
In this case, the DOS automatically knows that it is a relative file. This syntax, and the one shown in the above section, both allow either reading or writing to the file.

In order to read or write, you must, **before any operation**, position the file pointer to the correct record position.

FORMAT FOR POSITION COMMAND:

PRINT# file#, "P" CHR$(channel#) CHR$(rec# lo) CHR$(rec# hi)
or optionally as
PRINT# file#, "P" CHR$(channel#) CHR$(rec#lo) CHR$(rec#hi) CHR$(position)

EXAMPLES OF POSITION COMMAND:



The 2-byte format for the record number is needed because one byte can only hold 256 different numbers, and we can have over 700 records in the file. The rec# lo contains the least significant part of the address, and the rec# hi is the most significant part. This could be translated to the actual record number by the formula REC# = REC HI * 256 + REC LO.

Let's assume we have a mailing list. The list consists of 8 pieces of data, according to this chart:

Field Name	Length		
		state	2
first name	12	zip code	9
last name	15	phone number	10
address line 1	20		
address line 2	20		
city	12	TOTAL	100

This is how the record length is determined. We would probably want to allow an extra character in length for each field, to allow for separations; otherwise the INPUT# command would pick up a much longer piece of the file than needed, just like in sequential files. Therefore, we'll set up a file with a length of 108 characters per record. In the first record, we'll put the number 1, representing the largest record# used so far. Here is the program as described so far:

```
 10 OPEN 1, 8, 15
 20 OPEN 2, 8, 3, "0:MAILING LIST,L,"+CHR$(108)
 30 GOSUB 900
 40 PRINT# 1, "p" CHR$(3) (CHR$(1) CHR$(0) CHR$(1)
 50 GOSUB 900
 60 IF E=50 THEN PRINT#2, 1: GOTO 40
 70 INPUT# 2, X
300 STOP
900 INPUT# 1, E, B$, C, D
910 IF (E=50) OR (E < 20) THEN RETURN
920 PRINT A; B; C; D: STOP: RETURN
```

Error #50 which is checked in line 60 of the program is the RECORD NOT PRESENT error, which means that the record hadn't been created yet. Writing into the record will solve the proglem. This error condition must be watched carefully within your programs.

So far, all it does is create the file and the first record, but doesn't actually put any data in it. Below is a greatly expanded version of the program, to actually allow you to work with a mailing list where the records are coded by numbers.

MAILING LIST READ AND WRITE PROGRAM:

```
  5  A(1) = 12:A(2) = 15:A(3) = 20:A(4) = 20:A(5) = 12:A(6) = 2:A(7) = 9:A(8) = 10
 10  OPEN1,8,15:OPEN2,8,3,"0:Mailing List,1,"+CHR$(108):GOSUB900
 20  PRINT#1,"p"CHR$(3)CHR$(1) CHR$(0) CHR$(1):INPUT#2,X
 30  INPUT"Read, Write, or End";J$:IFJ$="e"THENCLOSE2:CLOSE1:END
 40  IFJ$="w"THEN200
 50  PRINT:INPUT"Record #";R:IFR<0ORR>XTHEN50
 60  IFR<2THEN30
 70  R1=R:R2=0:IFR1>256THENR2=INT(R1/256):R1=R1-256*R2
 80  RESTORE:DAT1,FIRST NAME,14,LAST NAME,30,ADDRESS1,51,ADDRESS2
 90  DATA72,CITY,85,STATE,88,ZIP,98,PHONE#
100  FORL=1TO8:READA,A$:PRINT#1,"p"CHR$(13) CHR$(R1) CHR$(R2) CHR$(A):GOSUB900
110  ONA/50GOTO50:INPUT#2,Z$:PRINTA$,Z$:NEXT:GOTO50
200  PRINT:INPUT"Record #";R:IFR<0ORR>5000THEN200
210  IFR<2THEN30
215  IFR>XTHENR=X+1:PRINT:PRINT"Record# "R
220  R1=R:R2=0:IFR1>256THENR2=INT(R1/256):R1=R1 - 256*R2
230  RESTORE:FORL=1TO8:READA,A$:PRINT#1, "p"CHR$(3) CHR$(R1) CHR$(R2) CHR$(A)
240  PRINTA$,:INPUTZ$:IFLEN(Z$)>A(L)THENZ$=LEFT$(Z$,A(L))
245  PRINT#2,Z$:NEXT:X=R:PRINT#1,"p"CHR$(3) CHR$(1) CHR$(0)
250  PRINT#2,X:GOTO200
900  INPUT#1,A,B$,C,D:IFA<20THENRETURN
910  IFA<>50THENPRINTA;B$,C;D:STOP:RETURN
920  IFJ$="r"THENPRINTB$
930  RETURN
```

This program asks for record numbers when retrieving records. It won't let you retrieve from beyond the end of the file, and if you try to write beyond the end it forces you to write on the next higher record.

A more advanced version than this would keep track of the items by "keys", to index the records. For example, you would probably want to search for a record by name, or print out labels by zip code. For this you would need a separate list of keys and record numbers, probably stored in sequential files.

When working with a new relative file that will soon be very large, it will save much time to create a record at the projected end of the file. In other words, if you expect the file to be 1000 records long, create a record# 1000 as soon as the file is created. This will force the DOS to create all intermediate records, making later use of those records much faster.

EXAMPLE OF CREATING LARGE FILE:

```
OPEN 1, 8, 15: OPEN 2, 8, 2, "REL,L,"+ CHR$(60)
PRINT# 1, "P" CHR$(2) CHR$(0) CHR$(4) CHR$(1)
PRINT# 2, "END"
CLOSE 2: CLOSE 1
```

8. PROGRAMMING THE DISK CONTROLLER

The expert programmer can actually design routines that reside and operate on the disk controller. DOS routines can be added that come from the diskette. Routines can be added much the same way as the DOS Support Program is "wedged" into your memory.

BLOCK-EXECUTE

This command will load a block from the diskette containing a machine language routine, and begin executing it at location 0 in the buffer until a RTS (ReTurn from Subroutine) command is encountered.

FORMAT FOR BLOCK-EXECUTE:

PRINT# file#, "BLOCK-EXECUTE:" channel, drive, track, block
or abbreviated as
PRINT# file#, "BLOCK-EXECUTE:" channel, drive, track, block

MEMORY-READ

There is 16K of ROM in the disk drive as well as 2K of RAM. You can get direct access to these, or to the buffers that the DOS has set up in the RAM, by using the MEMORY commands. MEMORY-READ allows you to select which byte to read, through the **error channel.**

FORMAT FOR MEMORY-READ:

PRINT# file#, "M-R" CHR$(low byte of address) CHR$(high byte)
(no abbreviation!)

The next byte read using the GET# statement through channel# 15, the error channel, will be from that address in the disk controller's memory, and successive bytes will be from successive memory locations.

Any INPUT# to the error channel will give peculiar results when you're using this command. This can be cleared up by any other command to the disk (besides a memory command).

PROGRAM TO READ THE DISK CONTROLLER'S MEMORY:

```
 10 OPEN 15, 8, 15
 20 INPUT "LOCATION PLEASE"; A
 30 A1= INT(A/256): A2= A- A1*256
 40 PRINT# 15, "M-R" CHR$(A2) CHR$(A1)
 50 FOR L=1 TO 5
 60 GET# 15, A$
 70 PRINT ASC(A$+ CHR$(0));
 80 NEXT
 90 INPUT "CONTINUE";A$
100 IF LEFT$(A$,1) ="Y" THEN 50
110 GOTO 20
```

MEMORY-WRITE

The MEMORY-WRITE command allows you to write up to 34 bytes at a time into the disk controller's memory. The MEMORY-EXECUTE and USER commands can be used to run this code.

FORMAT FOR MEMORY-WRITE:

PRINT# file#, "M-W" CHR$(low address byte) CHR$(high address byte)
#-of-characters; byte data

PROGRAM TO WRITE A "RTS" TO DISK:

```
10 OPEN 15, 8, 15, "M-W" CHR$(0) CHR$(5); 1; CHR$(96)
20 PRINT# 15, "M-E" CHR$(0) CHR$(19): REM JUMPS TO BYTE, RETURNS
30 CLOSE 15
```

MEMORY-EXECUTE

Any routine in the DOS memory, RAM or ROM, can be executed with the MEMORY-EXECUTE command.

FORMAT FOR MEMORY-EXECUTE:

PRINT# file#, "M-E" CHR$(low address byte) CHR$(high byte)

See line 20 above for an example.

USER Commands

Aside from the USER1 and USER2 commands discussed in chapter 6, and the UI+ and UI– commands in chapter 2, the USER commands are jumps to a table of locations in the disk drive's RAM memory.

USER COMMAND	FUNCTION
U1 or UA	BLOCK-READ without changing buffer-pointer
U2 or UB	BLOCK-WRITE without changing buffer-pointer
U3 or UC	jump to $0500
U4 or UD	jump to $0503
U5 or UE	jump to $0506
U6 or UF	jump to $0509
U7 or UG	jump to $050C
U8 or UH	jump to $050F
U9 or UI	jump to $FFFA
U; or UJ	power-up vector
UI+	set Commodore 64 speed
UI-	set VIC 20 speed

By loading these locations with another jump command, like JMP $0520, you can create longer routines that operate in the disk's memory along with an easy-to-use jump table–even from BASIC!

EXAMPLES OF USER COMMAMDS:

```
PRINT# 15, "U3"
PRINT# 15, "U"+ CHR$(50+Q)
PRINT# 15, "UI"
```

9. CHANGING THE DISK DRIVE DEVICE NUMBER

Software Method

The device number is selected by the drive by looking at a hardware jumper on the board and writing the number based on that jumper in a section of its RAM. Once operation is underway, it is easy to write over the previous device number with a new one.

FORMAT FOR CHANGING DEVICE NUMBER:

PRINT# file#, "M-W:" CHR$(119) CHR$(0) CHR$(2) CHR$(address+32) CHR$(address+64)

EXAMPLE OF CHANGING DEVICE NUMBER:

PRINT# 15, "M-W" CHR$(119) CHR$(0) CHR$(2) CHR$(9+32) CHR$(9+64)
PRINT# Q, "M-W" CHR$(119) CHR$(0) CHR$(2) CHR$(R+32) CHR$(R+64)

If you have more than one drive, it's sensible to change the address through hardware (see below). If you must, the procedure is easy. Just plug in the drives one at a time, and change their numbers to the desired new values. That way you won't have any conflicts.

Hardware Method

It's an easy job to permanently change the device number of your drive for use in multiple drive systems. The tools needed is a phillips-head screwdriver and a knife.

STEPS TO CHANGING DEVICE NUMBER ON HARDWARE:

1. Disconnect all cables from drive, including power.
2. Turn drive upside down on a flat, steady surface.
3. Remove 4 screws holding drive box together.
4. Carefully turn drive right side up, and remove case top.
5. Remove 2 screws on side of metal housing.
6. Remove housing.
7. Locate device number jumpers. If facing the front of the drive, it's on the left edge in the middle of the board.
8. Cut either or both of jumpers 1 and 2.
9. Replace housing and 2 screws, and case top and 4 screws.
10. Re-connect cables and power up.

The jumper number is added to the old device number (8) when cut. In other words, jumper 1 adds 1, and jumper 2 adds 2, to the device number. If none are cut, the number is 8, if 1 is cut it goes up to 9, and if only 2 is cut the number is 10. If both 1 and 2 are cut, the number is 11.

Appendix A: Disk Command Summary

General Format: PRINT# file#, command

COMMAND	COMMAND FORMAT
NEW	"N
COPY	"CO:new file=0:original file
RENAME	"R0:new name=0:old name
SCRATCH	"S0:file name
INITIALIZE	"I
VALIDATE	"V
DUPLICATE	not for single drives
BLOCK-READ	"B-R:" channel; drive; track; block
BLOCK-WRITE	"B-W:" channel; drive; track; block
BLOCK-ALLOCATE	"B-A:" drive; track; block
BLOCK-FREE	"B-F:" drive; track; block
BUFFER-POINTER	"B-P:" channel; position
USER1 and USER2	"Un:" channel; drive; track; block
POSITION	"P" CHR$(channel#) CHR$(rec# lo) CHR$(rec# hi) CHR$(position)
BLOCK-EXECUTE	"B-E:" channel; drive; track; block
MEMORY-READ	"M-R:" CHR$(address lo) CHR$(address hi)
MEMORY-WRITE	"M-W:" CHR$(address lo) CHR$(address hi) CHR$ (# chars) "data"
MEMORY-EXECUTE	"M-E:" CHR$(address lo) CHR$(address hi)
USER Commands	"Un:"

Appendix B: Summary of CBM Floppy Error Messages

0	OK, no error exists.
1	Files scratched response. Not an error condition.
2-19	Unused error messages: should be ignored.
20	Block header not found on disk.
21	Sync character not found.
22	Data block not present.
23	Checksum error in data.
24	Byte decoding error.
25	Write-verify error.
26	Attempt to write with write protect on.
27	Checksum error in header.
28	Data extends into next block.
29	Disk id mismatch.
30	General syntax error.
31	Invalid command.
32	Long line.
33	Invalid filename.
34	No file given.
39	Command file not found.
50	Record not present.
51	Overflow in record.
52	File too large.
60	File open for write.
61	File not open.
62	File not found.
63	File exists.
64	File type mismatch.
65	No block.
66	Illegal track or sector.
67	Illegal system track or sector.
70	No channels available.
71	Directory error.
72	Disk full or directory full.
73	Power up message, or write attempt with DOS Mismatch.
74	Drive not ready. (8050 only)

DESCRIPTION OF DOS ERROR MESSAGES

NOTE: Error message numbers less than 20 should be ignored with the exception of 01 which gives information about the number of files scratched with the SCRATCH command.

20: READ ERROR (block header not found)
The disk controller is unable to locate the header of the requested data block. Caused by an illegal sector number, or the header has been destroyed.

21: READ ERROR (no sync character)
The disk controller is unable to detect a sync mark on the desired track. Caused by misalignment of the read/write head, no diskette is present, or unformatted or improperly seated diskette. Can also indicate a hardware failure.

22: READ ERROR (data block not present)
The disk controller has been requested to read or verify a data block that was not properly written. This error message occurs in conjunction with the BLOCK commands and indicates an illegal track and/or sector request.

23: READ ERROR (checksum error in data block)
This error message indicates that there is an error in one or more of the data bytes. The data has been read into the DOS memory, but the checksum over the data is in error. This message may also indicate grounding problems.

24: READ ERROR (byte decoding error)
The data or header has been read into the DOS memory, but a hardware error has been created due to an invalid bit pattern in the data byte. This message may also indicate grounding problems.

25: WRITE ERROR (write-verify error)
This message is generated if the controller detects a mismatch between the written data and the data in the DOS memory.

26: WRITE PROTECT ON
This message is generated when the controller has been requested to write a data block while the write protect switch is depressed. Typically, this is caused by using a diskette with a write protect tab over the notch.

27: READ ERROR (checksum error in header)
The controller has detected an error in the header of the requested data block. The block has not been read into the DOS memory. This message may also indicate grounding problems.

28: WRITE ERROR (long data block)
The controller attempts to detect the sync mark of the next header after writing a data block. If the sync mark does not appear within a pre-determined time, the error message is generated. The error is caused by a bad diskette format (the data extends into the next block), or by hardware failure.

29: DISK ID MISMATCH
This message is generated when the controller has been requested to access a diskette which has not been initialized. The message can also occur if a diskette has a bad header.

30: SYNTAX ERROR (general syntax)
The DOS cannot interpret the command sent to the command channel. Typically, this is caused by an illegal number of file names, or patterns are illegally used. For example, two file names may appear on the left side of the COPY command.

31: SYNTAX ERROR (invalid command)
The DOS does not recognize the command. The command must start in the first position.

32: SYNTAX ERROR (long line)
The command sent is longer than 58 characters.

33: SYNTAX ERROR (invalid file name)
Pattern matching is invalidly used in the OPEN or SAVE command.

34: SYNTAX ERROR (no file given)
The file name was left out of a command or the DOS does not recognize it as such. Typically, a colon (:) has been left out of the command.

39: SYNTAX ERROR (invalid command)
This error may result if the command sent to command channel (secondary address 15) is unrecognizable by the DOS.

50: RECORD NOT PRESENT
Result of disk reading past the last record through INPUT#, or GET# commands. This message will also occur after positioning to a record beyond end of file in a relative file. If the intent is to expand the file by adding the new record (with a PRINT# command), the error message may be ignored. INPUT or GET should not be attempted after this error is detected without first repositioning.

51: OVERFLOW IN RECORD
PRINT# statement exceeds record boundary. Information is truncated. Since the carriage return which is sent as a record terminator is counted in the record size, this message will occur if the total characters in the record (including the final carriage return) exceeds the defined size.

52: FILE TOO LARGE
Record position within a relative file indicates that disk overflow will result.

60: WRITE FILE OPEN
This message is generated when a write file that has not been closed is being opened for reading.

61: FILE NOT OPEN
This message is generated when a file is being accessed that has not been opened in the DOS. Sometimes, in this case, a message is not generated; the request is simply ignored.

62: FILE NOT FOUND
The requested file does not exist on the indicated drive.

63: FILE EXISTS
The file name of the file being created already exists on the diskette.

64: FILE TYPE MISMATCH
The file type does not match the file type in the directory entry for the requested file.

65: NO BLOCK
This message occurs in conjunction with the B-A command. It indicates that the block to be allocated has been previously allocated. The parameters indicate the track and sector available with the next highest number. If the parameters are zero (0), then all blocks higher in number are in use.

66: ILLEGAL TRACK AND SECTOR
The DOS has attempted to access a track or sector which does not exist in the format being used. This may indicate a problem reading the pointer to the next block.

67: ILLEGAL SYSTEM T OR S
This special error message indicates an illegal system track or sector.

70: NO CHANNEL (available)
The requested channel is not available, or all channels are in use. A maximum of five sequential files may be opened at one time to the DOS. Direct access channels may have six opened files.

71: DIRECTORY ERROR
The BAM does not match the internal count. There is a problem in the BAM allocation or the BAM has been overwritten in DOS memory. To correct this problem, reinitialize the diskette to restore the BAM in memory. Some active files may be terminated by the corrective action.
NOTE: BAM = Block Availability Map

72: DISK FULL

Either the blocks on the diskette are used or the directory is at its limit of 152 entries for the 2040, 3040, and 4040 or 243 entries for the 8050. DISK FULL is sent when two blocks are available on the 8050 to allow the current file to be closed.

73: DOS MISMATCH (73, CBM DOS V2.6 1541)

DOS 1 and 2 are read compatible but not write compatible. Disks may be interchangeably read with either DOS, but a disk formatted on one version cannot be written upon with the other version because the format is different. This error is displayed whenever an attempt is made to write upon a disk which has been formatted in a non-compatible format. (A utility routine is available to assist in converting from one format to another.) This message may also appear after power up.

74: DRIVE NOT READY

An attempt has been made access the 1541 Single Drive Floppy Disk without any diskettes present in either drive.

APPENDIX C: Demonstration Disk Programs

1. DIR

```
4 OPEN2,8,15
5 PRINT"■":GOTO 10000
10 OPEN1,8,0,"$0"
20 GET#1,A$,B$
30 GET#1,A$,B$
40 GET#1,A$,B$
50 C=0
60 IF A$<>"" THEN C=ASC(A$)
70 IF B$<>"" THEN C=C+ASC(B$)*256
80 PRINT"■"MID$(STR$(C),2);TAB(3);"■";
90 GET#1,B$:IF ST<>0 THEN 1000
100 IF B$<>CHR$(34) THEN 90
110 GET#1,B$:IF B$<>CHR$(34)THEN PRINTB$;:GOTO110
120 GET#1,B$:IF B$=CHR$(32) THEN 120
130 PRINT TAB(18);:C$=""
140 C$=C$+B$:GET#1,B$:IF B$<>""  THEN 140
150 PRINT"■"LEFT$(C$,3)
160 GET T$:IF T$<>"" THEN GOSUB 2000
170 IF ST=0 THEN 30
1000 PRINT" BLOCKS FREE"
1010 CLOSE1:GOTO 10000
2000 IF T$="Q" THEN CLOSE1:END
2010 GET T$:IF T$="" THEN 2000
2020 RETURN
4000 REM DISK COMMAND
4010 C$="":PRINT">";
4011 GETB$:IFB$="" THEN4011
4012 PRINTB$;:IF B$=CHR$(13) THEN 4020
4013 C$=C$+B$:GOTO 4011
4020 PRINT#2,C$
5000 PRINT"■";
5010 GET#2,A$:PRINTA$;:IF A$<>CHR$(13)GOTO5010
5020 PRINT"■"
10000 PRINT "D-DIRECTORY"
10010 PRINT ">-DISK COMMAND"
10020 PRINT "Q-QUIT PROGRAM"
10030 PRINT "S-DISK STATUS "
10100 GETA$:IFA$=""THEN10100
10200 IF A$="D" THEN 10
10300 IF A$="." OR A$=">" OR A$="$" THEN 4000
10310 IF A$="Q" THEN END
10320 IF A$="S" THEN 5000
10999 GOTO 10100
```

2. VIEW BAM

```
100 REM ***********************************
101 REM *  VIEW BAM FOR VIC & 64 DISK  *
102 REM ***********************************
105 OPEN15,8,15
110 PRINT#15,"I0":NU$="N/A N/A N/A N/A N/A":Z4=1
120 OPEN2,8,2,"#"
130 Y$="■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■"
140 X$="■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■"
150 DEF FNS(Z) = 2↑(S-INT(S/8)*8) AND (SB(INT(S/8)))
```

```
160 PRINT#15,"U1:";2;0;18;0
170 PRINT#15,"B-P";2;1
180 PRINT"■";
190 Y=22:X=1:GOSUB430
200 FORI=0TO20:PRINT:PRINT"■■"RIGHT$(STR$(I)+" ",3);:NEXT
210 GET#2,A$
220 GET#2,A$
230 GET#2,A$
240 TS=0
250 FORT=1TO17:GOSUB450
260 Y=22:X=T+4:GOSUB430:GOSUB540:NEXT
270 FORI=1TO2000:NEXT:PRINT"■"
280 Y=22:X=1:GOSUB430
290 FORI=0TO20:PRINT:PRINT"■■"RIGHT$(STR$(I)+" ",3);:NEXT
300 FORT=18TO35
310 GOSUB450
320 Y=22:X=T-13:GOSUB430:GOSUB540:NEXT
330 FORI=1TO1000:NEXT
340 PRINT"■■■■■■"
350 PRINT#15,"B-P";2;144
360 N$="":FORI=1TO20:GET#2,A$:N$=N$+A$:NEXT
370 PRINT" "N$" "TS-17;"BLOCKS FREE"
380 FORI=1TO4000:NEXT
390 PRINT"■"
400 INPUT"■■■■■ANOTHER DISKETTE N■■■■";A$
410 IFA$="Y"THENRUN
420 IFA$<>"Y"THENEND
430 PRINTLEFT$(Y$,Y)LEFT$(X$,X)"■■";
440 RETURN
450 GET#2,SC$:SC=ASC(RIGHT$(CHR$(0)+SC$,1))
460 TS=TS+SC
470 GET#2,A$:IFA$=""THENA$=CHR$(0)
480 SB(0)=ASC(A$)
490 GET#2,A$:IFA$=""THENA$=CHR$(0)
500 SB(1)=ASC(A$)
510 GET#2,A$:IFA$=""THENA$=CHR$(0)
520 SB(2)=ASC(A$)
530 RETURN
540 PRINT"■■■"RIGHT$(STR$(T),1);"■■■";
550 REM  PRINTT"    "SC"  "SB(0)" "SB(1)" "SB(2)=CHR$(0)
560 IFT>24ANDS=1STHEN:PRINTMID$(NU$,Z4,1);:GOTO660
570 FORS=0TO20
580 IFT<1STHEN620
590 IFT>30ANDS=17THEN:PRINTMID$(NU$,Z4,1);:GOTO660
600 IFT>24ANDS=1STHEN:PRINTMID$(NU$,Z4,1);:GOTO660
610 IFT>24ANDS=19THENPRINTMID$(NU$,Z4,1);:GOTO660
620 IFT>17ANDS=20THENPRINTMID$(NU$,Z4,1);:Z4=Z4+1:GOTO660
630 PRINT"■";
640 IF FNS(S)=0 THEN PRINT"+";:GOTO660
650 PRINT"■+";:REMPIGHT$(STR$(S),1);Z4,1);:GOTO72
660 PRINT"■■";
670 NEXT
680 RETURN
```

3. DISPLAY T & S

```
100 REM********************************
110 REM* DISPLAY ANY TRACK $ SECTOR *
120 REM* ON THE DISK TO THE SCREEN  *
130 REM* OR THE PRINTER             *
140 REM********************************
150 PRINT"▯▯▯________________________"
160 PRINT"DISPLAY BLOCK CONTENTS"
165 PRINT"‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾":
170 REM********************************
180 REM* SET PROGRAM CONSTANT       *
190 REM********************************
200 SP$=" ":NL$=CHR$(0):HX$="0123456789ABCDEF"
210 FS$="":FORI=64 TO 95:FS$=FS$+"▯"+CHR$(I)+"■":NEXT I
220 SS$=" ":FOR I=192 TO 223:SS$=SS$+"▯"+CHR$(I)+"■":NEXT I
240 DIM A$(15),NB(2)
251 D$="0"
253 PRINT"          ▯S■CREEN▯▯▯▯▯▯▯▯▯OR ▯▯▯P■RINTER"
254 GETJJ$:IF JJ$="" THEN254
255 IF JJ$="S"THENPRINT"          ▯▯SCREEN■"
256 IF JJ$="P"THENPRINT"          ▯▯PRINTER■"
260 OPEN15,8,15,"I"+D$:GOSUB 650
265 OPEN4,4
270 OPEN 2,8,2,"#":GOSUB 650
280 REM********************************
290 REM* LOAD TRACK AND SECTOR      *
300 REM* INTO DISK BUFFER           *
310 REM********************************
320 INPUT"▯▯▯TRACK, SECTOR";T,S
330 IF T=0 OR T>35 THEN PRINT#15,"I"D$:CLOSE2:CLOSE4:CLOSE15:PRINT"END":END
340 IF JJ$="S" THEN PRINT"▯▯▯TRACK"T" SECTOR"S"▯"
341 IF JJ$="P" THEN PRINT#4:PRINT#4,"TRACK"T" SECTOR"S:PRINT#4
350 PRINT#15,"U1:2,"D$;T;S:GOSUB650
360 REM********************************
370 REM* READ BYTE 0 OF DISK BUFFER *
390 REM********************************
400 PRINT#15,"B-P:2,1"
410 PRINT#15,"M-R"CHR$(0)CHR$(5)
420 GET#15,A$(0):IFA$(0)=""THENA$(0)=NL$
428 IF JJ$="S"THEN430
430 IF JJ$="P"THEN460
431 REM********************************
432 REM* READ & CRT DISPLAY         *
433 REM* REST OF THE DISK BUFFER    *
434 REM********************************
436 K=1:NB(1)=ASC(A$(0))
438 FOR J=0 TO 63:IF J=32 THEN GOSUB 710:IF Z$="N"THEN J=80:GOTO 458
440 FOR I=K TO 3
442 GET#2,A$(I):IF A$(I)="" THEN A$(I)=NL$
444 IF K=1 AND I<2 THEN NB(2)=ASC(A$(I))
446 NEXT I:K=0
448 A$="":B$=":":N=J*4:GOSUB 790:A$=A$+":"
450 FOR I=0 TO 3:N=ASC(A$(I)):GOSUB 790
452 C$=A$(I):GOSUB 850:B$=B$+C$
454 NEXT I:IF JJ$="S" THEN PRINTA$B$
458 NEXT J:GOTO571
```

```
460 REM*********************************
462 REM* READ & PRINTER DISPLAY      *
464 REM*********************************
466 K=1:NB(1)=ASC(A$(0))
468 FOR J=0 TO 15
470 FOR I=K TO 15
472 GET#2,A$(I):IF A$(I)="" THEN A$(I)=NL$
474 IF K=1 AND I<2 THEN NB(2)=ASC(A$(I))
476 NEXT I:K=0
478 A$="":B$=":":N=J*16:GOSUB 790:A$=A$+":"
480 FOR I=0 TO 15:N=ASC(A$(I)):GOSUB 790:IF Z$="N"THEN J=40:GOTO 571
482 C$=A$(I):GOSUB 850:B$=B$+C$
484 NEXT I
486 IF JJ$="P" THEN PRINT#4,A$B$
488 NEXT J:GOTO571
571 REM*********************************
572 REM* NEXT TRACK AND SECTOR       *
573 REM*********************************
575 PRINT"NEXT TRACK AND SECTOR"NB(1)NB(2) "■"
580 PRINT"DO YOU WANT NEXT TRACK AND SECTOR"
590 GET Z$:IF Z$="" THEN590
600 IF Z$="Y" THEN T=NB(1):S=NB(2):GOTO330
610 IF Z$="N" THEN 320
620 GOTO 590
630 REM*********************************
640 REM* SUBROUTINES                 *
650 REM*********************************
660 REM* ERROR ROUTINE               *
670 REM*********************************
680 INPUT#15,EN,EM$,ET,ES:IF EN=0 THEN RETURN
690 PRINT"■DISK ERROR■"EN,EM$,ET,ES
700 END
710 REM*********************************
720 REM* SCREEN CONTINUE MESSAGE     *
730 REM*********************************
740 PRINT"■■■■■CONTINUE(Y/N)"
750 GETZ$:IF Z$="" THEN 750
760 IF Z$="N" THEN RETURN
770 IF Z$<>"Y" THEN 750
780 PRINT"□TRACK" T " SECTOR"S "□":RETURN
790 REM*********************************
800 REM* DISK BYTE TO HEX PRINT      *
810 REM*********************************
820 A1=INT(N/16):A$=A$+MID$(HX$,A1+1,1)
830 A2=INT(N-16*A1):A$=A$+MID$(HX$,A2+1,1)
840 A$=A$+SP$:RETURN
850 REM*********************************
860 REM* DISK BYTE TO ASC DISPLAY    *
870 REM* CHARACTER                   *
880 REM*********************************
890 IF ASC(C$)<32 THEN C$=" ":RETURN
910 IF ASC(C$)<128 OR ASC(C$)>159 THEN RETURN
920 C$=MID$(SS$,3*(ASC(C$)-127),3):RETURN
```

4. CHECK DISK

```
1 REM CHECK DISK -- VER 1.4
2 DN=8:REM FLOPPY DEVICE NUMBER
5 DIMT(100):DIMS(100):REM BAD TRACK, SECTOR ARRAY
9 PRINT"■■■■────────────────────────"
10 PRINT" CHECK DISK PROGRAM"
12 PRINT"────────────────────────"
20 D$="0"
30 OPEN15,DN,15
35 PRINT#15,"V"D$
45 N%=RND(TI)*255
50 A$="":FORI=1TO255:A$=A$+CHR$(255AND(I+N%)):NEXT
60 GOSUB900
70 OPEN2,DN,2,"#"
80 PRINT:PRINT#2,A$;
85 T=1:S=0
90 PRINT#15,"B-A:"D$;T;S
100 INPUT#15,EN,EM$,ET,ES
110 IFEN=0THEN130
115 IFET=0THEN200:REM END
120 PRINT#15,"B-A:"D$;ET;ES:T=ET:S=ES
130 PRINT#15,"U2:2,"D$;T;S
134 NB=NB+1:PRINT" CHECKED  BLOCKS"NB
135 PRINT" TRACK     ■■■■■"T;" SECTOR     ■■■■■"S"■"
140 INPUT#15,EN,EM$,ES,ET
150 IF EN=0THEN85
160 T(J)=T:S(J)=S:J=J+1
165 PRINT"■■BAD BLOCK:■■■",T;S""
170 GOTO85
200 PRINT#15,"I"D$
210 GOSUB900
212 CLOSE2
215 IFJ=0THENPRINT"■■■■■■■NO BAD BLOCKS!":END
217 OPEN2,DN,2,"#"
218 PRINT"■■BAD BLOCKS","TRACK","SECTOR"
220 FORI=0TOJ-1
230 PRINT#15,"B-A:";D$,T(I);S(I)
240 PRINT,,T(I),S(I)
250 NEXT
260 PRINT"■"J"BAD BLOCKS HAVE BEEN ALLOCATED"
270 CLOSE2:END
900 INPUT#15,EN,EM$,ET,ES
910 IF EN=0 THEN RETURN
920 PRINT"■■ERROR #"EN,EM$;ET;ES""
930 PRINT#15,"I"D$
```

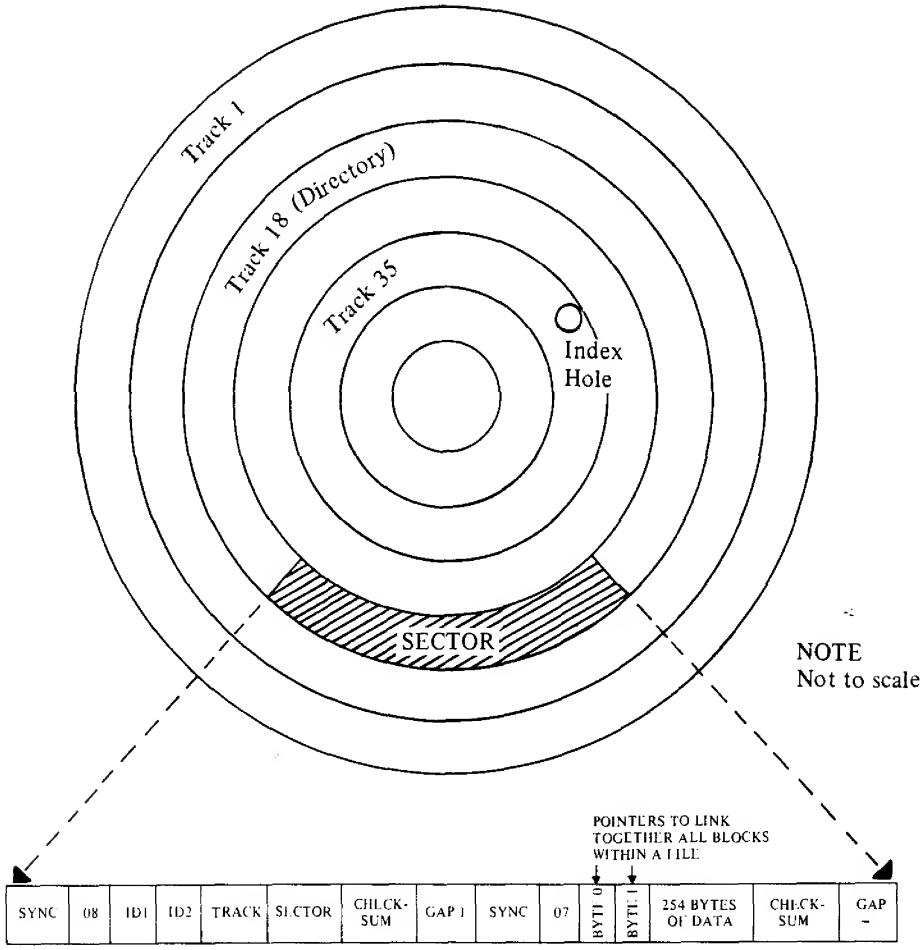
5. PERFORMANCE TEST

```
1000 REM  PERFORMANCE TEST  2.0
1010 :
1020 REM  VIC-20 AND COMMODORE 64
1030 REM  SINGLE FLOPPY DISK DRIVE
1040 :
1050 OPEN 1,8,15:OPEN15,8,15
1060 LT=35
1070 LT$=STR$(LT)
```

```
1080 NT=30
1090 PRINT"▯▮────────────────────────────"
1100 PRINT"    PERFORMANCE TEST"
1110 PRINT"────────────────────────────"
1120 PRINT
1130 PRINT"  INSERT SCRATCH"
1140 PRINT
1150 PRINT"     DISKETTE IN DRIVE"
1160 PRINT
1170 PRINT"▮    PRESS ▮RETURN▮"
1180 PRINT
1190 PRINT"              WHEN READY▮"
1200 FOR I=0 TO 50:GET A$:NEXT
1210 GET A$:IF A$<>CHR$(13) THEN 1210
1220 :
1230 :
1240 TI$="000000"
1250 TT=18
1260 PRINT#1,"N0:TEST DISK,00"
1270 C1$="   DISK NEW COMMAND    "+CHR$(13)
1280 C2$="▮ WAIT ABOUT 80 SECONDS"
1290 CC$=C1$+C2$:GOSUB 1840
1300 IF TI<NTTHEN1370
1310 PRINT"▮SYSTEM IS"
1320 PRINT"▮          NOT RESPONDING"
1330 PRINT" CORRECTLY TO COMMANDS"
1340 GOSUB 1880
1350 :
1360 :
1370 PRINT"▮DRIVE PASS"
1380 PRINT"          MECHANICAL TEST▮"
1390 TT=21
1400 OPEN 2,8,2,"0:TEST FILE,S,W"
1410 CC$="OPEN WRITE FILE"          :GOSUB 1840
1420 CH=2:CC$="WRITE DATA"          :GOSUB 1930
1430 CC$="CLOSE "+CC$               :GOSUB 1840
1440 OPEN 2,8,2,"0:TEST FILE,S,R"
1450 CC$="OPEN READ FILE"           :GOSUB 1840
1460 CH=2:GOSUB 1990
1470 PRINT#1,"S0:TEST FILE"
1480 CC$="SCRATCH FILE▮":TT=1       :GOSUB 1840
1490 :
1500 :
1510 TT=21
1520 OPEN 4,8,4,"#"
1530 NN%=(1+RND(TI)*254+NN%)AND255:PRINT#1,"B-P";4;NN%
1540 NN$="":FOR I=1 TO 255:NN$=NN$+CHR$(I):NEXT
1550 PRINT# 4,NN$;
1560 PRINT# 1,"U2:";4;0;LT;0
1570 CC$="WRITE TRACK"+LT$:GOSUB 1840
1580 PRINT#1,"U2:";4;0;1;0
1590 CC$="WRITE TRACK 1"            :GOSUB 1840
1600 PRINT#1,"U1:";4;0;LT;0
1610 CC$="READ TRACK"+LT$           :GOSUB 1840
1620 PRINT#1,"U1:";4;0;1;0
1630 CC$="READ TRACK 1"             :GOSUB 1840
1640 CLOSE 4
1650 :
1660 :
```

```
1670 PRINT"■ UNIT HAS PASSED"
1680 PRINT"     PERFORMANCE TEST!"
1690 PRINT"■ PULL DISKETTE FROM"
1700 PRINT"■  DRIVE BEFORE TURNING"
1710 PRINT"   POWER OFF."
1720 END
1730 :
1740 :
1750 PRINT"   ■CONTINUE (Y/N)?";
1760 FOR I=0 TO 50:GET A$:NEXT
1770 GET A$:IF A$="" THEN 1770
1780 PRINT A$"■"
1790 IF A$="N" THEN END
1800 IF A$="Y" THEN RETURN
1810 GOTO 1760
1820 :
1830 :
1840 PRINT CC$
1850 INPUT# 1,EN,EM$,ET,ES
1860 PRINTTAB(12)""EN;EM$;ET;ES;""
1870 IF EN<2 THEN RETURN
1880 PRINT"■ UNIT IS FAILING"
1890 PRINT"■   PERFORMANCE TEST"
1900 TM$=TI$:GOSUB 1750:TI$=TM$:RETURN
1910 :
1920 :
1930 PRINT"WRITING DATA"
1940 FOR I=1000 TO 2000:PRINT#CH,I:NEXT
1950 GOSUB1850
1960 CLOSE CH:RETURN
1970 :
1980 :
1990 PRINT"READING DATA"
2000 GETA$
2010 FOR I=1000 TO 2000
2020 INPUT# CH,J
2030 IF J<>I THEN PRINT"■READ ERROR:■":GOSUB 1850
2040 NEXT
2050 GOSUB 1850
2060 CLOSE CH:RETURN
```

APPENDIX D: DISK FORMATS



1540/1541 Format: Expanded View of a Single Sector

Block Distribution by Track

2040, 3040 Track number	Block or Sector Range	Total
1 to 17	0 to 20	21
18 to 24	0 to 19	20
25 to 30	0 to 17	18
31 to 25	0 to 16	17
4040 **Track number**	**Block or** **Sector Range**	**Total**
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17
8050 **Track number**	**Block or** **Sector Range**	**Total**
1 to 39	0 to 28	29
40 to 53	0 to 26	27
54 to 64	0 to 24	25
65 to 77	0 to 22	23

1540/1541 BAM FORMAT

Track 18, Sector 0.

BYTE	CONTENTS	DEFINITION
0,1	18,01	Track and sector of first directory block.
2	65	ASCII character A indicating 4040 format.
3	0	Null flag for future DOS use.
4-143		Bit map of available blocks for tracks 1-35.

*1 = available block
0=block not available
(each bit represents one block)

* STRUCTURE OF SINGLE DIRECTORY ENTRY

BYTE	CONTENTS	DEFINITION
0	128+type	File type OR'ed with $80 to indicate properly closed file. TYPES: 0 = DELeted 1 = SEQential 2 = PROGram 3 = USER 4 = RELative
1-2		Track and sector of 1st data block.
3-18		File name padded with shifted spaces.
19-20		Relative file only: track and sector for first side sector block.
21		Relative file only: Record size.
22-25		Unused.
26-27		Track and sector of replacement file when OPEN@ is in effect.
28-29		Number of blocks in file: low byte, high byte.

SEQUENTIAL FORMAT

BYTE	DEFINITION
0-1	Track and sector of next sequential data block.
2-256	254 bytes of data with carriage return as record terminators.

PROGRAM FILE FORMAT

BYTE	DEFINITION
0,1	Track and sector of next block in program file.
2-256	254 bytes of program info stored in CBM memory format (with key words tokenized). End of file is marked by three zero bytes.

1540/1541 DIRECTORY HEADER

Track 18, Sector 0.		
BYTE	CONTENTS	DEFINITION
144-161		Disk name padded with shifted spaces.
162-163		Disk ID.
164	160	Shifted space.
165-166	50,65	ASCII representation for 2A which is DOS version and format type.
166-167	160	Shifted spaces.
177-255	0	Nulls, not used.
Note: ASCII characters may appear in locations 180 thru 191 on some diskettes.		

DIRECTORY FORMAT

Track 18, Sector 1	
BYTE	DEFINITION
0-1	Track and sector of next directory block.
2-31	*File entry 1
34-63	*File entry 2
66-95	*File entry 3
98-127	*File entry 4
130-159	*File entry 5
162-191	*File entry 6
194-223	*File entry 7
226-255	*File entry 8

RELATIVE FILE FORMAT

DATA BLOCK	
BYTE	DEFINITION
0,1	Track and sector of next data block.
2-256	254 bytes of data. Empty records contain FF (all binary ones) in the first byte followed by 00 (binary all zeros) to the end of the record. Partially filled records are padded with nulls (00).
SIDE SECTOR BLOCK	
BYTE	DEFINITION
0-1	Track and sector of next side sector block.
2	Side sector number (0-5)
3	Record length
4-5	Track and sector of first side sector (number 0)
6-7	Track and sector of second side sector (number 1)
8-9	Track and sector of third side sector (number 2)
10-11	Track and sector of fourth side sector (number 3)
12-13	Track and sector of fifth side sector (number 4)
14-15	Track and sector of sixth side sector (number 5)
16-256	Track and sector pointers to 120 data blocks

Printed in Japan 1540031-02-830201

commodore
COMPUTER